Not All Hackers Are Evil: Automated ReDos Generator for Detecting Vulnerable Regexes

LENKA TUROŇOVÁ, Brno University of Technology, Czech Republic
LUKÁŠ HOLÍK, Brno University of Technology, Czech Republic
ONDŘEJ LENGÁL, Brno University of Technology, Czech Republic
MARGUS VEANES, Microsoft, USA
TOMÁŠ VOJNAR, Brno University of Technology, Czech Republic

We propose a solution to the problem of efficient matching regular expressions (regexes) with bounded repetition, such as `(ab){1,100}`, using deterministic automata. For this, we introduce novel *counting-set automata (CsAs)*, automata with registers that can hold sets of bounded integers and can be manipulated by a limited portfolio of constant-time operations. We present an algorithm that compiles a large sub-class of regexes to deterministic CsAs. This includes (1) a novel Antimirov-style translation of regexes with counting to *counting automata (CAs)*, nondeterministic automata with bounded counters, and (2) our main technical contribution, a determinization of CAs that outputs CsAs. The main advantage of this workflow is that the size of the produced CsAs does not depend on the repetition bounds used in the regex (while the size of the DFA is exponential to them). Our experimental results confirm that deterministic CsAs produced from practical regexes with repetition are indeed vastly smaller than the corresponding DFAs. More importantly, our prototype matcher based on CsA simulation handles practical regexes with repetition regardless of sizes of counter bounds. It easily copes with regexes with repetition where state-of-the-art matchers struggle.

CCS Concepts: • **Theory of computation** → **Regular languages**; *Quantitative automata*; • **Security and privacy** → **Denial-of-service attacks**; • **Applied computing** → *Document searching*.

Additional Key Words and Phrases: regular epxression matching, bounded repetition, ReDos, determinization, Antimirov's derivatives, counting automata, counting-set automata

ACM Reference Format:
Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Margus Veanes, and Tomáš Vojnar. 2020. Not All Hackers Are Evil: Automated ReDos Generator for Detecting Vulnerable Regexes . *Proc. ACM Program. Lang.* 4, OOPSLA, Article 218 (November 2020), 27 pages. https://doi.org/10.1145/3428286

1 INTRODUCTION

Matching *regexes* (regular expressions) is a ubiquitous component of software, used, e.g., for searching, data validation, parsing, finding and replacing, data scraping, or syntax highlighting. It is commonly used and natively supported in most programming languages [?]. For instance,

Authors' addresses: Lenka Turoňová, Faculty of Information Technology, Brno University of Technology, Božetěchova 2, Brno, 612 00, Czech Republic, ituronova@fit.vutbr.cz; Lukáš Holík, Faculty of Information Technology, Brno University of Technology, Božetěchova 2, Brno, 612 00, Czech Republic, holik@fit.vutbr.cz; Ondřej Lengál, Faculty of Information Technology, Brno University of Technology, Božetěchova 2, Brno, 612 00, Czech Republic, lengal@fit.vutbr.cz; Margus Veanes, MSR, Microsoft, One Microsoft Way, Redmond, 98052, USA, margus@microsoft.com; Tomáš Vojnar, Faculty of Information Technology, Brno University of Technology, Božetěchova 2, Brno, 612 00, Czech Republic, vojnar@fit.vutbr.cz.

2475-1421/2020/11-ART218
https://doi.org/10.1145/3428286

about 30–40 % of Java, JavaScript, and Python software use regex matching (as reported in multiple studies, see, e.g., [?]).

The efficiency of regex matching engines has a significant impact on the overall usability of software applications. Unpredictability of a matcher's performance may lead to catastrophic consequences, witnessed by events such as the recent catastrophic outage of Cloudflare services [?], caused by a single poorly written regex, and it is a doorway for the so-called ReDoS attack, a denial of service attack based on overwhelming a regex matching engine by providing a specially crafted regex or text. For instance, in 2016, ReDoS caused an outage of StackOverflow [?] or rendered vulnerable websites that used the popular Express.js framework [?]. Works such as [??] give arguments that ReDoS is not just a niche problem but rather a common and serious threat.

Failures of matching are mostly caused by the so-called "catastrophic backtracking", a situation when variants of Spencer's simulation of a *nondeterministic finite automaton* (NFA) by *backtracking* [?] exhibit a behaviour super-linear to the length of the text. Matching algorithms based on backtracking are probably the most often implemented ones, their performance is, however, at worst *exponential* to the text length. An alternative with a much lower worst-case complexity (wrt the length of the text) is to use *deterministic finite automata* (DFAs). In the ideal case, the DFA is pre-computed; matching can then be linear to the text length, with each input symbol processed in constant time. This is the so-called *static DFA simulation* [?]. The major drawback of static DFA simulation is that the DFA construction may explode, rendering the method unusable in practice.

Variants of Thompson's algorithm [?] (sometimes called NFA simulation or NFA-to-DFA simulation) avoid the explosion by working directly with the NFA. They essentially run the determinization by subset construction *on the fly*, always remembering only the current DFA state. On reading a character, a successor DFA state is computed and used to replace the current state. The disadvantage of Thompson's algorithm is that, for a highly nondeterministic NFA, the DFA states—sets of the states of the NFA—may get large and computing a DFA-state successor over a symbol becomes expensive, linear to the size of the NFA (compared to the constant time of static DFA simulation).

Modern matchers therefore use caching of already visited parts of the DFA. Making a step within the cached part is then as fast as with the explicitly determinized automaton. Extremely efficient implementations of Thompson's algorithm with caching are used in `RE2` [?] and GNU `grep` [?]. Their close cousin, an on-the-fly Brzozowski's derivative construction, is implemented in the tool `SRM` [?]. Highly nondeterministic regexes[1] that lead to exploding determinization are, however, problematic for all variants, explicit determinization as well as NFA simulation, with or without caching.

In this paper, we focus on eliminating a frequent cause of a DFA explosion—a use of the *counting operator*, also known as the operator of *bounded repetition*. It succinctly expresses repeated patterns such as `(ab){1,100}`, representing 1 to 100 consecutive repetitions of `ab`. Such expressions are very common (cf. [?]), e.g., in the RegExLib library [?], which collects expressions for recognizing URIs, markup code, pieces of Java code, or SQL queries; in the Snort rules [?] used for detecting attacks in network traffic; in real-life XML schemas, with the counter bounds being as large as 10 million [?]; or in detecting information leakage from traffic logs [?].

To illustrate the principal difficulty with matching bounded repetitions, especially when combined with a high degree of nondeterminism, consider the regex `.*a.{`k`}` where $k \in \mathbb{N}$ (the regex denotes strings where the symbol `a` appears k positions from the end of the word). Already the NFA will have at least k states, which is exponential to the regex size because k is written in decimal. Due

[1] Loosely speaking, a "highly nondeterministic regex" is one for which the determinization of the NFA created by some of the usual algorithms explodes. Determinism of regexes closely corresponds to the notion of *1-unambiguity of the regex* [??]: when matching a text from left to right against the regex, it is always clear which letter of the regex matches the text character.

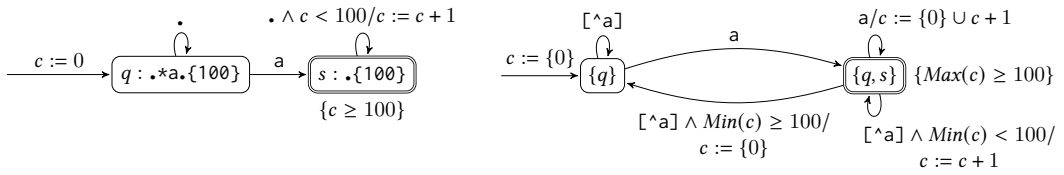
to the inherent nondeterminism of this regex, determinization then adds a second level of the exponential explosion. Indeed, the minimal DFA accepting the language has 2^{k+1} states because it must remember all the positions where the symbol a was seen during the last $k+1$ steps. This requires a finite memory of $k+1$ bits and thus 2^{k+1} reachable DFA states. Determinizing the NFA explicitly is thus out of question for even moderate values of k. The pure Thompson's NFA simulation is feasible but very slow, as reading each character may in the worst case require processing the entire NFA. Moreover, caching of the DFA state space, used in industrial matchers like `RE2` [?] or GNU `grep` [?], may also be ineffective due to the size of the state space and low cache utilization. At the same time, combinations of nondeterminism and counting are fairly common. A high degree of nondeterminism is, for instance, usual when searching for a pattern "anywhere on the line" (corresponding to prefixing the pattern with `.*`), which is the standard behaviour for GNU `grep` and similar programs when start/end of line anchors are not used.

To facilitate efficient matching of such nondeterministic counting, we propose a translation from regexes with repetition to deterministic machines that are succinct and can perform matching with nearly constant character complexity. The novel succinct and fast deterministic machine, called the *counting-set automaton (CsA)*, is the key to the result. It is a deterministic finite automaton with a special type of registers that can hold values called *counting sets*—a set of bounded integer values—and support a limited selection of simple set operations. Crucially for the efficiency of our approach, we show that, using a suitable data structure, all the set operations can be implemented to run in *constant time* regardless of the size of the set.

Our compilation from regexes to CsAs proceeds in two steps. First, we compile the regexes into nondeterministic *counting automata (CAs)*, automata with counters whose values are *a priori* bounded. Variants of CAs have been used in several other works under different names, e.g., [???????]. The compilation from regexes is cheap and produces automata whose size is independent of the counter bounds and linear in the size of the regex. We present a novel translation of regexes to CAs that generalizes the Antimirov's derivative construction [?]. Our translation has several advantages over the existing alternatives, such as absence of ϵ-transitions in the output CA and succinctness. The result of translating the regex `.*a.`$\{k\}$ into a CA is illustrated in Fig. 1a.

The main step forward we make in this paper is a solution of efficient matching for a large class of highly nondeterministic regexes with counting that are quite common in practice. The main technical problem we have solved is a succinct transformation of a (nondeterministic) CA into a *deterministic CsA*. Our algorithm produces a CsA in *time independent of the counter bounds*. We note that this has been a known open problem (emphasized, e.g., in [?]). Works on matching of bounded repetition such as [???????] mostly focus on deterministic regexes and do not propose practical solutions for the nondeterministic case. We have carried out an extensive experimental evaluation of our algorithm on a large sample of regexes used for pattern matching in various applications. The experiments show that our algorithm, although also limited to a sub-class of regexes, handles over 90 % of regexes with counting we collected. The obtained data confirm that our CsAs are indeed far smaller and can be constructed faster than corresponding DFAs. Most importantly, we demonstrate the practical relevance of our algorithm for pattern matching. We have implemented a matching algorithm based on CsA simulation[2] and compared it with several state-of-the-art matchers, namely, `grep` [?], `RE2` [?], `SRM` [?], and `.NET` [?]. Our results show that problematic highly nondeterministic regexes with counting indeed appear in practice and can also be easily crafted as a ReDoS attack, and that CsAs can efficiently solve most of such problematic cases. For instance, the regex `(_a){64999}_a` from [?] can cause state-of-the-art matchers exceed

[2]We use a pre-computed deterministic CsA. While on-the-fly determinization is also possible, it was not needed in our experimentation since we have not witnessed problems with CsA state space explosion.



(a) The CA for the regex .*a.{k}

(b) The CsA from determinization of (a)

Fig. 1. The nondeterministic CA and the deterministic CsA for .*a.{100}. The transitions are labeled by their *guard*, which gives the character class (in the standard POSIX regex notation, where, e.g., . stands for "any character") and possibly restricts counter values, delimited by "/" from the counter *update*. If a counter does not have the update specified, then the transition does not change its value. In (b), the notation $c + 1$ stands for the set of values obtained by incrementing each value in c and then *removing* values larger than the upper bound 100 of the counter. The edges denoting initial states are labelled with *initial values* of the counters. Final states are labelled with an *acceptance condition*, e.g. $\{c \geq 100\}$ in (a).
The formal counter operations OP_c presented later in Section 4 are in (a) shown as follows: the guard of OP_c is shown in conjunction with the character guard α on the left of the "/", the update of OP_c is shown on the right of "/" in the form of an assignment to c, where INCR_c appears as the right value $c + 1$, EXIT as 0, EXIT1 as 1, and NOOP is omitted.

any reasonable time limit (when searching for the pattern anywhere on the line, with the implicit .* in front). Already with the counter bounds lowered to 1,000, the matchers take from 6 to 34 seconds on 500 KiB of text, but our algorithm needs only 1 second even with the original bound 64,999.

We summarize the technical contributions of this work as follows:

(1) A novel Antimirov style regex-to-CA translation.
(2) A novel notion of the counting-set automaton, a deterministic machine that allows for succinct representation of counting constraints and fast matching.
(3) CA-to-CsA determinization that runs in time independent of counter bounds.
(4) A regex-matching algorithm interconnecting the above, efficient regardless of counter bounds especially on regexes that combine counting with nondeterminism.
(5) Implementation and extensive experimental evaluation of the above.

2 OVERVIEW

We give a brief overview of our conversion of a regex with counting into a deterministic CsA. We use the example regex R = .*a.{100}, discussed already in the introduction and representing strings where the symbol a appears 100 positions from the end, with the corresponding minimum DFA having 2^{101} states. The conversion proceeds in two steps. First, R is translated into a nondeterministic CA (Fig. 1a), denoted as CA(R); second, CA(R) is converted into a deterministic CsA (Fig. 1b). The size of both is independent of the counter bounds (both of the automata will have 2 states only).

Counting-Set Data Structure. Before looking into the conversion from regular expressions to CsAs it is useful to first understand *why* the resulting CsA can be used efficiently for matching in the first place. The main enabler behind this is the use of our *counting set* data structure, say c, representing sets $S_c \subseteq \{0, \ldots, \boldsymbol{max}_c\}$ where the upper bound $\boldsymbol{max}_c$ is a fixed positive integer. A *runtime value* of c is a tuple (o, ℓ) where $o \in \mathbb{N}$ is called an *offset* and ℓ is a queue of strictly increasing natural numbers such that $S_c = \{o - n \mid n \in \ell\}$.

The data structure supports *constant-time* implementations of the following operations, assuming the access to the first and the last element of the queue in constant time (the queue may be implemented as a doubly linked list).

- The minimum and the maximum of S_c are obtained as $o - \mathit{last}(\ell)$ and $o - \mathit{first}(\ell)$, respectively.
- Insert 0: if $o - \mathit{last}(\ell) > 0$, then append o at the end of ℓ (similarly for inserting 1)
- Increment all, up to $\boldsymbol{max}_c$: $o := o + 1$; if $o - \mathit{first}(\ell) > \boldsymbol{max}_c$, then remove $\mathit{first}(\ell)$.
- Reset to $\{0\}$: $\ell := 0$; $o := 0$ (similarly for reset to $\{1\}$).

The independence of the run-time of these operations from $\boldsymbol{max}_c$ enables our major achievement:

the independence of the running time from the counter bounds.

Let us now illustrate how this data structure works during matching. We run the CsA in Fig. 1b, assuming the meaning of the operations provided above, over the sample input word $\texttt{aa0}^{(10)}\texttt{aab}^{(86)}\texttt{dfa}$.

The configurations of the automaton after processing prefixes of the word are shown in the table (the control state, the counting-set run-time configuration (o, ℓ) and the value S_c it represents. The state $\{q, s\}$ fulfills the *accepting condition* after processing the 6th and the 7th prefix, since the maximum of S_c at these points is indeed at least 100.

prefix	state	(o, ℓ)	S_c
ϵ	$\{q\}$	$(0, [0])$	$\{0\}$
a	$\{q, s\}$	$(0, [0])$	$\{0\}$
aa	$\{q, s\}$	$(1, [0, 1])$	$\{1, 0\}$
$\texttt{aa0}^{(10)}$	$\{q, s\}$	$(11, [0, 1])$	$\{11, 10\}$
$\texttt{aa0}^{(10)}\texttt{aa}$	$\{q, s\}$	$(13, [0, 1, 12, 13])$	$\{13, 12, 1, 0\}$
$\texttt{aa0}^{(10)}\texttt{aab}^{(87)}$	$\{q, s\}$	$(100, [0, 1, 12, 13])$	$\{100, 99, 88, 87\}$
$\texttt{aa0}^{(10)}\texttt{aab}^{(86)}\texttt{d}$	$\{q, s\}$	$(101, [1, 12, 13])$	$\{100, 89, 88\}$
$\texttt{aa0}^{(10)}\texttt{aab}^{(86)}\texttt{df}$	$\{q, s\}$	$(102, [12, 13])$	$\{89, 88\}$
$\texttt{aa0}^{(10)}\texttt{aab}^{(86)}\texttt{dfa}$	$\{q, s\}$	$(103, [12, 13, 103])$	$\{90, 89, 0\}$

From a Nondeterministic CA to a Deterministic CsA. The idea of our CA-to-CsA determinization is best explained by comparison with the naive determinization of a CA, which would create a DFA by the explicit textbook-style subset construction. The states of the DFA would then be sets of runtime configurations of the CA where each CA-configuration would consist of a control state and a counter valuation. Counter valuations would hence be "unfolded"—they would become an explicit part of the DFA control states—and the succinctness provided by counters would be lost. For instance, the run of the CA in Fig. 1a on the word aaa... generates "powerstates"

$$\{(q, c{=}0)\},\ \{(q, c{=}0), (s, c{=}0)\},\ \{(q, c{=}0), (s, c{=}0), (s, c{=}1)\},\ \{(q, c{=}0), (s, c{=}0), (s, c{=}1), (s, c{=}2)\}, \ldots$$

which are essentially subsets of $\{q, s\} \times \{0, \ldots, 100\}$. In the worst case, the size of the DFA would be exponential in counter bounds because s can be paired with any subset of $\{0, \ldots, 100\}$ recording possible values of c. In contrast to this, as illustrated above, our CsA represents the counter valuations implicitly: it computes them dynamically on the fly and stores them as *counting sets*—i.e., the valuation of a counter changes from an integer to a *set* of integers. The counter valuations are hence not a part of control states, and their overall number influences neither the size of the CsA nor the time needed to build them.

Fig. 1b shows the CsA obtained from determinization of the CA in Fig. 1a. The runtime configurations of the CsA, reached for the word aaa are

$$(\{q\}, c{\in}\{0\}),\quad (\{q, s\}, c{\in}\{0\}),\quad (\{q, s\}, c{\in}\{0, 1\}),\quad (\{q, s\}, c{\in}\{0, 1, 2\}),$$

which precisely corresponds to the first three steps of the sample DFA run above. Namely, the control states are kept in the first component and the counter values are in the second component, i.e., the set S_c represented by the run-time values of c. The second is not relevant for the states

where the counter is never active (always 0; in this case q). The implicit value 0 of c in q is not recorded in the counting sets.

We note that some DFA powerstates cannot be encoded as CsA configurations due to the involved Cartesian abstraction: essentially, any state in the powerset is paired with any counter value from the counting set. Hence, our approach does not handle the full class of regexes. Fortunately, as our empirical evidence shows, regexes that fall out from the supported class are rare in practice.

From Regexes to Nondeterministic CAs. To translate a regex into a CA, we propose a generalization of the Antimirov's derivative construction [?] to symbolic counting. In Antimirov's setting, a derivative of a regex R wrt a character class α is a set of regexes that together capture all tails of words in $\mathcal{L}(R)$ whose head character is from α. In particular, according to [?], which generalizes [?] to *explicit* counting, the derivatives of the regex R = `.*a.{100}` wrt the classes `a` and `[^a]` are $\{R, \texttt{.\{100\}}\}$ and $\{R\}$, respectively. Further, for $1 \leq k \leq 100$, the derivative of `.{`k`}` wrt both `a` and `[^a]` is $\{\texttt{.\{}k-1\texttt{\}}\}$. The derivatives become the states of the resulting NFA, with R being initial and `.{0}` final, and with α-transitions from each regex to all its α-derivatives (for α being either `a` or `[^a]`). The obtained NFA is already large, it has 102 states.

In our new construction, the counting is kept *implicit* using symbolic counters. Instead of modifying the counter bound of the derivative (by, e.g., deriving `.{99}` from `.{100}`), we keep the original bound unchanged and use a counter c to keep track of the difference between the original value and the current value. Our *conditional derivative* operator $\partial_\alpha(\cdot)$ then equips the produced derivatives with *conditional counter updates* to keep the counters up-to-date. For instance, $\partial_{\texttt{a}}(\texttt{.\{100\}})$ returns the same regex `.{100}`, but it is paired with conditional counter updates for c, namely, "if $c < 100$, then increment c; and if $c \geq 100$, then exit the counting loop". The CA we obtain this way is shown in Fig. 1a, where the first conditional update translates to the self loop on the state `.{100}` and the second to the acceptance condition. The size of the CA does not depend on the counter bounds.

3 PRELIMINARIES

We cast our definitions in the framework of symbolic automata [?], a natural succinct representations of finite-state transition relations over large alphabets of labels. Symbolic automata work over alphabets equipped with a so-called effective Boolean algebra, which defines the needed interface for handling large sets of labels on automata transitions.

Before providing the formal definition of an effective Boolean algebra, we start with an intuitive example, which is also going to be the alphabet algebra used throughout the paper, including the experiments. Later on, we will further leverage the general definition to work with algebras over counter and counting-set predicates.

Example 3.1. Regular expressions in practice use *character classes* as basic building blocks. To simplify the discussion, let us restrict our attention to ASCII as the character universe $\mathfrak{D}$. In other words, $\mathfrak{D}$ is the set $\{n \mid 0 \leq n < 2^7\}$ of all 7-bit characters represented using their character codes. Then, for example, the character classes `[0-9]` and `[A-Z]` denote, respectively, the set $[\![\texttt{[0-9]}]\!] = \{n \mid 48 \leq n \leq 57\}$ of all digit codes, and the set $[\![\texttt{[A-Z]}]\!] = \{n \mid 65 \leq n \leq 90\}$ of all upper-case letter codes. Character classes made up of individual symbols such as `@` denote singleton sets, e.g., $[\![\texttt{@}]\!] = \{64\}$. Character classes can also be used to form *unions*, they can be *complemented*, and even *subtracted* from each other, and are in general closed under Boolean operations. There are therefore many different ways how to represent the same character sets, e.g., $[\![\texttt{[0-9]}]\!] = [\![\texttt{[0-45-9]}]\!] = [\![\texttt{[0-4]}]\!] \cup [\![\texttt{[5-9]}]\!]$. To illustrate the complement, for example, `[^0-9]` denotes the set of all non-digits, as does `[\x00-\x2F\x3A-\x7F]`. The set of all character classes is then an example of the set Ψ of all *predicates* of a Boolean algebra, and checking *satisfiability* of a predicate $\varphi \in \Psi$ means to decide whether φ denotes a *nonempty* set. For example, the predicate `[]`

is unsatisfiable because $[\![\texttt{[]}]\!] = \emptyset$ and $\texttt{.}$ denotes the *true* predicate because $[\![\texttt{.}]\!] = \mathfrak{D}$. Further, note that a character class can, without loss of generality, be represented as a Boolean combination of *intervals* or even as a union of intervals if normalized. □

3.1 Effective Boolean Algebras

An *effective Boolean algebra* $\mathbb{A}$ has components $(\mathfrak{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ where $\mathfrak{D}$ is a *universe* of underlying domain elements. Ψ is a set of unary *predicates* closed under the Boolean connectives $\vee, \wedge : \Psi \times \Psi \to \Psi$ and $\neg : \Psi \to \Psi$; and $\bot, \top \in \Psi$ are the *false* and *true* predicates. Values of the algebra are sets of domain elements, and the *denotation function* $[\![\_]\!] : \Psi \to 2^{\mathfrak{D}}$ satisfies that $[\![\bot]\!] = \emptyset$, $[\![\top]\!] = \mathfrak{D}$, and for all $\varphi, \psi \in \Psi$, $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$, $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$, and $[\![\neg\varphi]\!] = \mathfrak{D} \setminus [\![\varphi]\!]$. For $\varphi \in \Psi$, we write $\boldsymbol{Sat}(\varphi)$ when $[\![\varphi]\!] \neq \emptyset$, and we say that φ is *satisfiable*. We require that $\boldsymbol{Sat}$ as well as $\vee$, $\wedge$, and $\neg$ are *computable* as a part of the definition of an effective Boolean algebra. We write $x \models \varphi$ for $x \in [\![\varphi]\!]$ and we use $\mathbb{A}$ as a subscript of a component when it is not clear from the context, e.g., $[\![\_]\!]_{\mathbb{A}} : \Psi_{\mathbb{A}} \to 2^{\mathfrak{D}_{\mathbb{A}}}$.

3.2 Words and Regexes

The basic building blocks of regexes are *predicates* from an effective Boolean algebra *CharClass* of *character classes*, such as the class of digits, written as \d. Let $\mathfrak{D} = \mathfrak{D}_{CharClass}$. A *word* over $\mathfrak{D}$ is a sequence of symbols $a_1 \cdots a_n \in \mathfrak{D}^*$ and a *language* $\mathcal{L}$ over $\mathfrak{D}$ is a subset of $\mathfrak{D}^*$. We use ϵ to denote the *empty word*. The concatenation of words u and v is denoted as $u \cdot v$ (often abbreviated to uv) and is lifted to sets as usual. We call $a \in \mathfrak{D}$ the *head* of the word $a.w$ and $w \in \mathfrak{D}^*$ its *tail*. Furthermore, we write $\mathcal{L}^n$ for the n-th power of $\mathcal{L} \subseteq \mathfrak{D}^*$ with $\mathcal{L}^0 \stackrel{\text{def}}{=} \{\epsilon\}$ and $\mathcal{L}^{n+1} \stackrel{\text{def}}{=} \mathcal{L}^n \cdot \mathcal{L}$.

The abstract syntax of regexes is the following with $\alpha \in \Psi_{CharClass}$ and n, m being integers such that $0 \leq n$, $0 < m$, and $n \leq m$:

$$\varepsilon \quad \alpha \quad R_1 \cdot R_2 \quad R_1|R_2 \quad R\{n,m\} \quad R*$$

where $R_1 \cdot R_2$ is called a *concat node* and $R_1|R_2$ is called a *choice node*. The semantics of a regex R is defined as a subset of $\mathfrak{D}^*$ in the following way: $\mathcal{L}(\alpha) \stackrel{\text{def}}{=} [\![\alpha]\!]$, $\mathcal{L}(\varepsilon) \stackrel{\text{def}}{=} \{\epsilon\}$, $\mathcal{L}(R_1R_2) \stackrel{\text{def}}{=} \mathcal{L}(R_1) \cdot \mathcal{L}(R_2)$, $\mathcal{L}(R_1|R_2) \stackrel{\text{def}}{=} \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$, $\mathcal{L}(R\{n,m\}) \stackrel{\text{def}}{=} \bigcup_{i=n}^{m} (\mathcal{L}(R))^i$, and $\mathcal{L}(R*) \stackrel{\text{def}}{=} \mathcal{L}(R)^*$. R is *nullable* if $\epsilon \in \mathcal{L}(R)$. We will also need to refer to the number of *character-class leaf nodes* of a regex R, denoted by $\sharp_\Psi(R)$ and defined as follows: $\sharp_\Psi(\varepsilon) = 0$, $\sharp_\Psi(\alpha) = 1$, $\sharp_\Psi(R_1 \cdot R_2) = \sharp_\Psi(R_1|R_2) = \sharp_\Psi(R_1) + \sharp_\Psi(R_2)$, $\sharp_\Psi(R\{n,m\}) = \sharp_\Psi(R*) = \sharp_\Psi(R)$.

3.3 Minterms

Let $Preds(R)$ be the set of all predicates that occur in a regex R, and let $Minterms(R)$ denote the set of *minterms* of $Preds(R)$. Intuitively, $Minterms(R)$ is a set of non-overlapping predicates that can be treated as a concrete finite alphabet. Each minterm is essentially a region in the Venn diagram of the predicates in R: it is a satisfiable conjunction $\bigwedge_{\psi \in Preds(R)} \psi'$ where $\psi' \in \{\psi, \neg\psi\}$. For example, if $R =$ `[0-z]{4}[0-8]{5}`, then $Preds(R) =$ {`[0-8]`, `[0-z]`} and $Minterms(R) =$ {`[0-8]`, `[9-z]`, `[^0-z]`}. Formally, if $\alpha \in Minterms(R)$, then $\boldsymbol{Sat}(\alpha)$ and $\forall \psi \in Preds(R)$: $[\![\alpha]\!] \cap [\![\psi]\!] \neq \emptyset \Rightarrow [\![\alpha]\!] \subseteq [\![\psi]\!]$.

Note that although the number of minterms of a general set X of predicates may be exponential in $|X|$, it is only linear if X consists of intervals of symbols used in regexes, such as `[a-zA-Z]` or `[^a-zA-Z]` (the former denotes two intervals while the latter their complement, which is equivalent to the union of three intervals). Intervals of numbers generate only a linear number of minterms.

3.4 Symbolic Automata

We use *symbolic finite automata (FAs)*, whose alphabet is given by an effective Boolean algebra, as a generalization of classical finite automata. Formally, an FA is a tuple $A = (\mathbb{I}, Q, q_0, F, \Delta)$ where $\mathbb{I}$ is

an effective Boolean algebra called the input algebra, Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $\Delta \subseteq Q \times \Psi_{\mathbb{I}} \times Q$ is a finite set of transitions. A transition $(q, \alpha, r) \in \Delta$ will be also written as $q\text{-}(\alpha)\!\!\rightarrow\! r$.

A *run of A from a state p_0* over a word $a_1 \cdots a_n$ is a sequence of transitions $(p_{i-1}\text{-}(\alpha_i)\!\!\rightarrow\! p_i)_{i=1}^{n}$ with $a_i \in [\![\alpha_i]\!]$; the run is *accepting* if $p_n \in F$. The *language of a state q*, denoted $\mathcal{L}_A(q)$, is the set of words over which A has an accepting run from q. The *language of A*, denoted $\mathcal{L}(A)$, is $\mathcal{L}_A(q_0)$. A classical finite automaton can be understood as an FA where the basic predicates have singleton set semantics, i.e., when for each concrete letter a there is a predicate α_a such that $[\![\alpha_a]\!] = \{a\}$. A is *deterministic* iff for all $p \in Q$ and all transitions $p\text{-}(\alpha)\!\!\rightarrow\! q$ and $p\text{-}(\alpha')\!\!\rightarrow\! r$, it holds that if $\alpha \wedge \alpha'$ is satisfiable, then $q = r$.

4 COUNTING AUTOMATA

Counting automata (CAs) are a natural and compact automata counterpart for regexes with counting. They are essentially a limited sub-class of classical counter automata, in which counters are only supposed to count the number of passes through some of its parts (corresponding to a counted sub-expression of a regex) and guards on transitions enforce a specified number of repetitions of that part before the automaton is allowed to move on.

Counter algebra. A *counter algebra* is an effective Boolean algebra $\mathbb{C}$ associated with a finite set C of counters. The counters play the role of bounded loop variables associated with a *lower bound* $\boldsymbol{min}_c \geq 0$ and an *upper bound* $\boldsymbol{max}_c > 0$ such that $\boldsymbol{min}_c \leq \boldsymbol{max}_c$. $\mathfrak{D}_{\mathbb{C}}$ is the set of interpretations $\mathfrak{m} : C \to \mathbb{N}$, called *counter memories* such that $0 \leq \mathfrak{m}(c) \leq \boldsymbol{max}_c$ for all $c \in C$. $\Psi_{\mathbb{C}}$ contains Boolean combinations of *basic* predicates $\textsc{canExit}_c$ and $\textsc{canIncr}_c$, for $c \in C$, whose semantics is given by

$$\mathfrak{m} \models \textsc{canExit}_c \iff \mathfrak{m}(c) \geq \boldsymbol{min}_c, \quad \mathfrak{m} \models \textsc{canIncr}_c \iff \mathfrak{m}(c) < \boldsymbol{max}_c.$$

Counting automata. A *counting automaton (CA)* is a tuple $A = (\mathbb{I}, C, Q, q_0, F, \Delta)$ where $\mathbb{I}$ is an effective Boolean algebra called the *input algebra*, C is a finite set of *counters* with an associated counter algebra $\mathbb{C}$, Q is a finite set of *states*, $q_0 \in Q$ is the *initial state*, $F : Q \to \Psi_{\mathbb{C}}$ is the *final-state condition*, and $\Delta \subseteq Q \times \Psi_{\mathbb{I}} \times (C \to \boldsymbol{O}) \times Q$ is the (finite) *transition relation*, where $\boldsymbol{O} = \{\textsc{exit}, \textsc{incr}, \textsc{exit1}, \textsc{noop}\}$ is the set of *counter operations*. The component f of a transition $p\text{-}(\alpha,f)\!\!\rightarrow\! q \in \Delta$ is its *(counter) operator*. We often view f as a set of *indexed operations* $\textsc{op}_c$ to denote the operation assigned to the counter c, $f(c) = \textsc{op}$.

Semantics of CAs. The semantics of the CA A is defined through its *configuration automaton $FA(A)$*, an FA whose states are A's *configurations*, i.e., pairs $(q, \mathfrak{m}) \in Q \times \mathfrak{D}_{\mathbb{C}}$ consisting of a state q and a counter memory $\mathfrak{m}$. To define $FA(A)$, we must first define the semantics of counter operators f, which occur on transitions. For this, we associate with each (indexed) operation $\textsc{op}_c$ a counter guard $grd(\textsc{op}_c)$ and a counter update $upd(\textsc{op})$ as shown on the right. Intuitively, the operation NOOP does not modify the counter's value and is always enabled. The operation INCR increments the counter and is enabled if the counter has not yet reached its upper bound. The operation EXIT resets the counter to 0 on exit from the counting loop and is enabled when the counter reaches its lower bound. The operation EXIT1 executes EXIT followed by INCR. The *guard* of a counter operator $f : C \to \boldsymbol{O}$ is then a predicate $\varphi_f \in \Psi_{\mathbb{C}}$ over counter memories, and its *update* $\mathrm{f} : \mathfrak{D}_{\mathbb{C}} \cup \{\bot\} \to \mathfrak{D}_{\mathbb{C}} \cup \{\bot\}$ is a counter-memory transformer:

$grd(\textsc{noop}_c) \stackrel{\text{def}}{=} \top_{\mathbb{C}}$		$upd(\textsc{noop}) \stackrel{\text{def}}{=} \boldsymbol{\lambda} n.n$	
$grd(\textsc{incr}_c) \stackrel{\text{def}}{=} \textsc{canIncr}_c$		$upd(\textsc{incr}) \stackrel{\text{def}}{=} \boldsymbol{\lambda} n.n+1$	
$grd(\textsc{exit}_c) \stackrel{\text{def}}{=} \textsc{canExit}_c$		$upd(\textsc{exit}) \stackrel{\text{def}}{=} \boldsymbol{\lambda} n.0$	
$grd(\textsc{exit1}_c) \stackrel{\text{def}}{=} \textsc{canExit}_c$		$upd(\textsc{exit1}) \stackrel{\text{def}}{=} \boldsymbol{\lambda} n.1$	

$$\varphi_f \stackrel{\text{def}}{=} \bigwedge_{\textsc{op}_c \in f} grd(\textsc{op}_c) \qquad \mathrm{f}(\mathfrak{m}) \stackrel{\text{def}}{=} \begin{cases} \boldsymbol{\lambda} c.upd(f(c))(\mathfrak{m}(c)) & \text{if } \mathfrak{m} \models \varphi_f \\ \bot & \text{otherwise} \end{cases}$$

Intuitively, f updates all counters in a counter memory $\mathfrak{m}$ by their corresponding operations if $\mathfrak{m}$ satisfies the guard, otherwise the result is $\bot$.

We now define the *configuration automaton* of A, denoted as $FA(A)$, which defines the language semantics of the CA A. The states of $FA(A)$ are the configurations of A (there are finitely many of them), and the initial state of $FA(A)$ is the *initial configuration* $(q_0, \{c \mapsto 0 \mid c \in C\})$ of A. A state $(p, \mathfrak{m})$ of $FA(A)$ is *final* iff $\mathfrak{m} \models F(p)$. The transition relation $\Delta_{FA(A)}$ of $FA(A)$ is defined as

$$\Delta_{FA(A)} = \{(p, \mathfrak{m}) \text{ -}(\alpha)\!\to (q, \mathrm{f}(\mathfrak{m})) \mid p \text{ -}(\alpha,f)\!\to q \in \Delta, \mathfrak{m} \models \varphi_f\}.$$

Deterministic and simple CAs. A is *deterministic* iff the following holds for every state $p \in Q$ and every two transitions $p\text{-}(\alpha_1,f_1)\!\to q_1$, $p\text{-}(\alpha_2,f_2)\!\to q_2 \in \Delta$: if both $\alpha_1 \wedge \alpha_2$ and $\varphi_{f_1} \wedge \varphi_{f_2}$ are satisfiable, then $f_1 = f_2$ and $q_1 = q_2$. It follows from the definitions that, if A is deterministic, then $FA(A)$ is deterministic too. A is *simple* if for any two transitions $q\text{-}(\alpha,f)\!\to r$ and $q'\text{-}(\alpha',f')\!\to r'$, either $\alpha = \alpha'$ or $[\![\alpha]\!] \cap [\![\alpha']\!] = \emptyset$. That is, different character guards do not overlap and can be mostly treated as plain symbols. We also require that all guards are satisfiable. CAs constructed from regexes by the algorithm in Section 5 will be simple.

Example 4.1. Fig. 1a shows a CA in an intuitive notation, with the initial state q and final conditions $F(q) = \bot, F(s) = \textsc{exit}_c$, where $\boldsymbol{min}_c = \boldsymbol{max}_c = 100$. The same notation is used in Fig. 2. Fig. 3a shows a CA in a notation following the formal development more closely. □

5 FROM A REGEX TO A CA VIA CONDITIONAL PARTIAL DERIVATIVES

We introduce a generalization of the Antimirov's partial derivative construction [?] to *symbolic* counting, which allows one to replace a verbose NFA by a succinct CA. The difference between the older variant of [?] with *explicit* counting [?] and our new version was already illustrated in Section 2. To recall it briefly using the example of the regex `.{100}`: from 100 partial derivatives $\partial_{\bullet}(\texttt{.\{i\}}) = \texttt{.\{i-1\}}$, $1 \le i \le 100$, and an NFA with 100 states and transitions $\texttt{.\{i\}}\text{-}(\bullet)\!\to\texttt{.\{i-1\}}$, the new construction will take us to the single derivative $\partial_{\bullet}(\texttt{.\{100\}}) = \{\texttt{.\{100\}}\}$ associated with a conditional counter update which induce an NFA with a single state and the transition $\texttt{.\{100\}}\text{-}(\alpha,\textsc{incr}_c)\!\to\texttt{.\{100\}}$.

We apply the construction on regexes that are normalized using the below rules where $X \rightsquigarrow Y$ denotes that X is rewritten to Y:

- All nested concat nodes are rewritten to the flattened right-associative *list form*, which is always maintained throughout the construction, using the rules: $(X \cdot Y) \cdot Z \rightsquigarrow X \cdot (Y \cdot Z)$, $\varepsilon \cdot Z \rightsquigarrow Z$, and $Z \cdot \varepsilon \rightsquigarrow Z$.
- If S is *nullable*, then $S\{\ell, k\} \rightsquigarrow S\{0, k\}$. Moreover, in the nullable context $S\{0, k\}$, S can be considered as if it was not nullable.

Observe that the normalization does not increase the size of the regex (it may decrease the size).

Let R be a fixed normalized regex. A subexpression of R that is of the form $X = S\{\ell, k\}$ is called a *counting loop*. We consider each counting loop to represent a *counter* whose name is the counting loop itself and whose *upper bound* is $\boldsymbol{max}_X = k$ and *lower bound* is $\boldsymbol{min}_X = \ell$. For example, $(\texttt{.\{9\}})*$ contains the counter $X = \texttt{.\{9\}}$ with $\boldsymbol{min}_X = \boldsymbol{max}_X = 9$. In the following, let C stand for the set of all counters that occur in R, also denoted by $\mathit{Counters}(R)$.

We use the convention that the juxtaposition XY of normalized regexes X and Y is again a normalized regex of the equivalent concat node $X \cdot Y$: e.g., if $X = a \cdot b$ and $Y = (a \cdot b)*$, then $XY = a \cdot (b \cdot (a \cdot b)*)$. Observe in particular that $X\varepsilon = X$. In other words, we treat concatenated elements as sequences, and a singleton sequence equals to the element itself.

Our construction will work over the set $\Sigma = \mathit{Minterms}(R)$ of minterms of R and produce simple CA that use minterms of Σ on transitions.

5.1 Parametric Languages

We define the language of a normalized regex starting with a counting loop relative to a counter value. For that, we lift the definition of languages to be parametric in counter memories $\mathfrak{m}$, but regexes other than the above are treated as before with the memory $\mathfrak{m}$ passed through unchanged.

$$\mathrm{L}^{\mathfrak{m}}(\varepsilon) \stackrel{\text{def}}{=} \{\epsilon\} \quad (1)$$

$$\mathrm{L}^{\mathfrak{m}}(\psi Z) \stackrel{\text{def}}{=} [\![\psi]\!] \cdot \mathrm{L}^{\mathfrak{m}}(Z) \quad (2)$$

$$\mathrm{L}^{\mathfrak{m}}((W|Y)Z) \stackrel{\text{def}}{=} \mathrm{L}^{\mathfrak{m}}(WZ) \cup \mathrm{L}^{\mathfrak{m}}(YZ) \quad (3)$$

$$\mathrm{L}^{\mathfrak{m}}(S{*}Z) \stackrel{\text{def}}{=} \mathrm{L}^{\mathfrak{m}}(S)^{*} \cdot \mathrm{L}^{\mathfrak{m}}(Z) \quad (4)$$

$$\mathrm{L}^{\mathfrak{m}}(S\{\ell,k\}Z) \stackrel{\text{def}}{=} \mathrm{L}^{\mathfrak{m}}(S) \cdot \mathrm{L}^{\textsc{incr}_{S\{\ell,k\}}(\mathfrak{m})}(S\{\ell,k\}Z) \cup \mathrm{L}^{\textsc{exit}_{S\{\ell,k\}}(\mathfrak{m})}(Z) \quad (5)$$

$$\mathrm{L}^{\perp}(X) \stackrel{\text{def}}{=} \emptyset \quad (\text{for all } X) \quad (6)$$

Recall that if f is a counter operator and $\mathfrak{m}$ a counter memory, then $\mathrm{f}(\mathfrak{m})$ denotes the appropriately updated memory where $\mathrm{f}(\mathfrak{m}) = \perp$ when f is not enabled in $\mathfrak{m}$. Below, if there is a single counter $c \in C$ such that $f(c) \neq \textsc{noop}$, we sometimes identify f with $\textsc{op}_c$ and use $\textsc{op}_c(\mathfrak{m})$ to represent the updated memory $\mathrm{f}(\mathfrak{m})$. Specifically, $\textsc{incr}_X$ (if enabled) increments the counter value of X by 1, and $\textsc{exit}_X$ (if enabled) resets the counter value of X to 0. Let $\mathfrak{m}$ be a counter memory. Then Cases (1)–(6) define the *parametric languages* of regexes. The intuition behind Case (4) is that all counters that may be present in S are inactive on the level of $S{*}$. Note also that Case (5) is well-defined since, for $X = S\{\ell,k\}$ and $\mathfrak{m}' = \textsc{incr}_X(\mathfrak{m})$, $k - \mathfrak{m}'(X) < k - \mathfrak{m}(X)$ if $\mathfrak{m}(X) < k$, and $\mathfrak{m}' = \perp$ if $\mathfrak{m}(X) = k$.

Let $0 \stackrel{\text{def}}{=} \boldsymbol{\lambda} c.0$ denote the initial memory that maps all counters to 0. The following below, proven in [?], relates $\mathrm{L}^{\mathfrak{m}}(R)$ with the non-parametric definition of regular languages.

Theorem 5.1. *Let R be a normalized regex. Then $\mathrm{L}^{0}(R) = \mathcal{L}(R)$.*

5.2 Conditional Derivation

We will now introduce our conditional derivative construction formally.

A *partial conditional derivative* is a pair $\langle f, X\rangle$ where f is a counter operator and X a normalized regex. Given a counter memory $\mathfrak{m}$, we let $\langle f, X\rangle$ define the language $\mathrm{L}^{\mathfrak{m}}(\langle f, X\rangle) \stackrel{\text{def}}{=} \mathrm{L}^{\mathrm{f}(\mathfrak{m})}(X)$. In other words, f is first applied to the counter memory $\mathfrak{m}$ and then the regex is evaluated in the updated memory. If f is not enabled in $\mathfrak{m}$, then the denoted language is empty.

A *conditional derivative* is a finite set of partial conditional derivatives. The language defined by a conditional derivative D in a counter memory $\mathfrak{m}$ is defined as the union of the languages of the partial conditional derivatives in D: $\mathrm{L}^{\mathfrak{m}}(D) \stackrel{\text{def}}{=} \bigcup_{d \in D} \mathrm{L}^{\mathfrak{m}}(d)$.

To define how conditional derivatives of a given regex looks like, we need a notion of a *sequential composition* of conditional derivatives $D \otimes E \stackrel{\text{def}}{=} \{\langle f; g, X{\cdot}Y\rangle \mid \langle f, X\rangle \in D, \langle g, Y\rangle \in E, f; g \neq \perp\}$ where $f; g \neq \perp$ is the composed counter operator such that $\mathrm{f}; \mathrm{g}(\mathfrak{m}) = \mathrm{g}(\mathrm{f}(\mathfrak{m}))$. The case when $f; g = \perp$ is discussed later on.

Conditional derivatives of a normalized regex are defined as shown on the right assuming that concatenations $X{\cdot}Y$ are normalized to the list form explained above, $\alpha \in \Sigma$, $\mathbf{ID}$ denotes the identity function $\boldsymbol{\lambda} x.x$, and $X = S\{\ell,k\}$ is a counting loop. Observe that, in $\partial_\alpha(S) \otimes \{\langle \textsc{incr}_X, XZ\rangle\}$, the operation $\textsc{incr}_X$ gets composed with $\textsc{noop}_X$, yielding $\textsc{incr}_X$ again, because $S\{\ell,k\}$ cannot occur in S. It is possible that in $\{\langle \textsc{exit}_X, \varepsilon\rangle\} \otimes \partial_\alpha(Z)$, X is in scope of Z (e.g., Z starts with X). The composition can then contain the operation $\textsc{exit}_X; \textsc{incr}_X$ that corresponds to $\textsc{exit1}_X$ because $\textsc{incr}_X$ is trivially enabled when the counter value of X is 0. The only other possible composition of individual operations that can appear in this case is $\textsc{exit}_X; \textsc{exit}_X$. If

$$\partial_\alpha(\varepsilon) \stackrel{\text{def}}{=} \emptyset$$

$$\partial_\alpha(\psi Z) \stackrel{\text{def}}{=} \begin{cases} \{\langle \mathbf{ID}, Z\rangle\} & \text{if } \alpha \wedge \psi \text{ is satisfiable} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\partial_\alpha((W|Y)Z) \stackrel{\text{def}}{=} \partial_\alpha(WZ) \cup \partial_\alpha(YZ)$$

$$\partial_\alpha(S{*}Z) \stackrel{\text{def}}{=} \partial_\alpha(S) \otimes \{\langle \mathbf{ID}, S{*}Z\rangle\} \cup \partial_\alpha(Z)$$

$$\partial_\alpha(XZ) \stackrel{\text{def}}{=} \partial_\alpha(S) \otimes \{\langle \textsc{incr}_X, XZ\rangle\} \cup$$

$\boldsymbol{min}_X = 0$, $\textsc{exit}_X; \textsc{exit}_X = \textsc{exit}_X$, which is well-defined because $\textsc{exit}_X$ is always enabled for $\boldsymbol{min}_X = 0$. If $\boldsymbol{min}_X > 0$, then $\textsc{exit}_X; \textsc{exit}_X$ is undefined, and $\textsc{exit}_X; \textsc{exit}_X$ does not contribute anything to the composition. However, this is correct since X is not nullable, and the second $\textsc{exit}_X$ is not enabled after the counter value of X is reset to 0. Intuitively, the second occurrence of X cannot be exited without iterating X at least once.

Example 5.2. Consider the regex $R =$ `.*a{1,3}a{1,3}a`. Let X be the counting loop `a{1,3}`. R has two minterms `a` and `[^a]`. We get the following conditional derivatives of R, starting with the case for $\partial_\alpha(S{*}Z)$ due to the normal form assumption:

$$\begin{aligned}
\partial_{\mathsf{a}}(R) &= \partial_{\mathsf{a}}(\mathbf{.}) \otimes \{\langle \mathbf{ID}, R\rangle\} \cup \partial_{\mathsf{a}}(XX\mathsf{a}) \\
&= \{\langle \mathbf{ID}, R\rangle, \langle \textsc{incr}_X, XX\mathsf{a}\rangle, \langle \textsc{exit1}_X, X\mathsf{a}\rangle\} \\
\partial_{\mathsf{a}}(XX\mathsf{a}) &= \partial_{\mathsf{a}}(\mathsf{a}) \otimes \{\langle \textsc{incr}_X, XX\mathsf{a}\rangle\} \cup \{\langle \textsc{exit}_X, \varepsilon\rangle\} \otimes \partial_{\mathsf{a}}(X\mathsf{a}) \\
&= \{\langle \textsc{incr}_X, XX\mathsf{a}\rangle\} \cup \{\langle \textsc{exit}_X, \varepsilon\rangle\} \otimes \{\langle \textsc{incr}_X, X\mathsf{a}\rangle, \langle \textsc{exit}_X, \varepsilon\rangle\} \\
&= \{\langle \textsc{incr}_X, XX\mathsf{a}\rangle, \langle \textsc{exit1}_X, X\mathsf{a}\rangle\} \\
\partial_{\mathsf{a}}(X\mathsf{a}) &= \partial_{\mathsf{a}}(\mathsf{a}) \otimes \{\langle \textsc{incr}_X, X\mathsf{a}\rangle\} \cup \{\langle \textsc{exit}_X, \varepsilon\rangle\} \otimes \partial_{\mathsf{a}}(\mathsf{a}) \\
&= \{\langle \textsc{incr}_X, X\mathsf{a}\rangle, \langle \textsc{exit}_X, \varepsilon\rangle\} \\
\partial_{\mathsf{a}}(\mathsf{a}) = \partial_{\mathsf{a}}(\mathbf{.}) = \partial_{\texttt{[\^{}a]}}(\mathbf{.}) &= \{\langle \mathbf{ID}, \varepsilon\rangle\} \\
\partial_{\texttt{[\^{}a]}}(\mathsf{a}) &= \emptyset
\end{aligned}$$

Above, the composition $\textsc{exit}_X; \textsc{exit}_X$ in $\partial_{\mathsf{a}}(XX\mathsf{a})$ is undefined and thus removed. We also get that $\partial_{\texttt{[\^{}a]}}(R) = \{\langle \mathbf{ID}, R\rangle\}$ where $\partial_{\texttt{[\^{}a]}}(\mathsf{a}) = \emptyset$ and consequently $\partial_{\texttt{[\^{}a]}}(XX\mathsf{a}) = \emptyset$ and $\partial_{\texttt{[\^{}a]}}(X\mathsf{a}) = \emptyset$.

If we now consider, for example, the language defined by $\partial_{\mathsf{a}}(X\mathsf{a})$ in a valid counter memory $\mathfrak{m}$, it is the union of the languages $\mathrm{L}^{\textsc{incr}_X(\mathfrak{m})}(X\mathsf{a})$ and $\mathrm{L}^{\textsc{exit}_X(\mathfrak{m})}(\varepsilon)$. These correspond to the cases of continuing to iterate the loop X (if the counter value of X is below 3) or exiting the loop (if the counter value of X is at least 1) and accepting $\{\epsilon\}$. □

Example 5.3. Consider the regex `(.{9})*`, whose CA is in Fig. 2. Here, `.` is the only input predicate and denotes the set of all characters. We explain the use of some of the counter operations in the CA of Fig. 2 by showing how they arise through the partial-derivative-based construction of CAs as discussed above. The initial state is the regex itself. The (only) partial derivative of the state `(.{9})*` is `.{9}(.{9})*` where the body of the counting loop is exited but also incremented once, so EXIT1 is applied to c under the guard $\textsc{canExit}_c$ (which is shown as $c \geq 9/c{:=}1$ in the figure). The state `.{9}(.{9})*` has two cases of partial derivatives both leading back to `.{9}(.{9})*`.

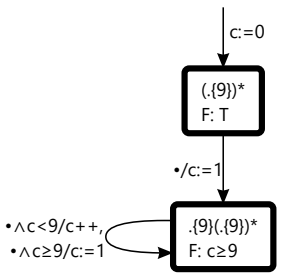


Fig. 2. CA((.{9})*)

The first case is when $c < 9$ ($\textsc{canIncr}_c$ holds), in which case c is incremented (shown as $c < 9/c{+}{+}$ in the figure). The second case is when the counting loop is conditionally nullable and is exited under the condition $\textsc{canExit}_c$ (i.e. $c \geq 9$), the value of c is reset to 0, and then c is incremented as a result of taking the partial derivative of `(.{9})*`. Thus, EXIT1 arises as a sequential composition of exiting the loop, followed by resetting the counter to 0, and then incrementing it. Therefore, $\textsc{canExit}_c$ must hold, while the increment condition holds trivially after a reset to 0.

The initial state is unconditionally final in Fig. 2, while the other state is final only when $\textsc{canExit}_c$ holds as marked by "F :". □

We now state the correctness theorem of conditional derivatives. For that, we define $\textsc{canExit}_R$ as the predicate shown above for a normalized regex R, assuming that X stands for a counting loop.

$$\textsc{canExit}_R \stackrel{\text{def}}{=} \begin{cases} \top_{\mathbb{C}} & \text{if } R = \varepsilon, \\ \textsc{canExit}_Z & \text{else if } R = YZ \text{ and } Y \text{ is nullable}, \\ \textsc{canExit}_X \wedge \textsc{canExit}_Z & \text{else if } R = XZ, \\ \bot_{\mathbb{C}} & \text{otherwise.} \end{cases}$$

Note that Y above may also be a counting loop. However, since it is nullable, $\boldsymbol{min}_Y$ must be 0, and then $\textsc{canExit}_Y$ is always true. (If $\boldsymbol{min}_Y > 0$, then Y cannot be nullable as R is normalized.)

We further need the following additional notions too. A counter X *is visible in* R if either $R = YZ$ and $X = Y$, or else if X does not occur in Y and X is visible in Z. A counter memory $\mathfrak{m}$ is *valid for* R if $\mathfrak{m}(X) = 0$ for all invisible counters X that occur in R. Correctness of the construction of conditional derivatives is stated in Theorem 5.4—see [?] for a detailed proof.

Theorem 5.4. *Let R be a normalized regex and let $\Sigma = Minterms(\Theta)$ where Θ is some finite superset of $Preds(R)$. If $\mathfrak{m}$ is valid for R, then $\mathrm{L}^{\mathfrak{m}}(R) = \bigcup_{\alpha\in\Sigma}[\![\alpha]\!] \cdot \mathrm{L}^{\mathfrak{m}}(\partial_\alpha(R)) \cup \{\epsilon \mid \mathfrak{m} \models \textsc{canExit}_R\}$.*

5.3 Constructing CAs from Conditional Derivatives

We convert a normalized regex R to the counting automaton $\mathrm{CA}(R)$ whose set of states is the smallest set containing R as the initial state and all those regexes that arise in conditional derivatives constructed from R by repeated derivation wrt Σ. Given a state represented by a regex S, for each $\alpha \in \Sigma$ and each partial conditional derivative $\langle f, T\rangle \in \partial_\alpha(S)$, there is a transition S-(α,f)→T in $\mathrm{CA}(R)$. The *final condition* $F(S)$ of a state S of $\mathrm{CA}(R)$ is $\textsc{canExit}_S$. Observe that $F(S) = \bot_{\mathbb{C}}$ when S is not nullable and has no visible counters, which corresponds to the classical case.

As shown in [?] the following result can be proved using Theorem 5.4.

Theorem 5.5. *Let R be a normalized regex and $A = FA(CA(R))$. Then, for all $\langle \mathfrak{m}, S\rangle \in Q_A$, $\mathcal{L}_A(\langle \mathfrak{m}, S\rangle) = \mathrm{L}^{\mathfrak{m}}(S)$.*

The construction of $\mathrm{CA}(R)$ terminates, and the number of states of $\mathrm{CA}(R)$ is linear in $\sharp_\Psi(R)$.

Theorem 5.6. *Let R be a normalized regex. Then $|Q_{CA(R)}| \leq \sharp_\Psi(R) + 1$.*

A proof of Theorem 5.6 is in [?]. We get the following final correctness result as a corollary of Theorem 5.5, Theorem 5.1, and Theorem 5.6.

Corollary 5.7. *Let R be a normalized regex. Then $\mathcal{L}(R) = \mathcal{L}(CA(R))$.*

Proof. First, $Q_{\mathrm{CA}(R)}$ is finite, and thus well-defined by using Theorem 5.6. Use Theorem 5.5 with $\langle \mathfrak{m}, S\rangle$ as the initial state $\langle 0, R\rangle$ of A. It follows that $\mathcal{L}(A) = \mathrm{L}^0(R)$. Then use Theorem 5.1 for $\mathrm{L}^0(R) = \mathcal{L}(R)$ and $\mathcal{L}(\mathrm{CA}(R)) = \mathcal{L}(A)$ holds by definition. □

A further important aspect of $\mathrm{CA}(R)$ is that, although the number of input minterms may potentially be exponential in the number of predicates in R, in the case of predicates being represented as a finite union of intervals (as is done typically for character classes), the size of a single predicate representation can be estimated to be proportional to the number of interval borders in the union. In this case, the total size of all the minterms remains linear in the total size of all the predicates because the total number of interval borders will remain the same in minterms as in the original set of predicates. In other words, mintermization based on character classes does not blow up the number of transition in $\mathrm{CA}(R)$. We have also validated this fact experimentally.

6 FROM COUNTING AUTOMATA TO COUNTING-SET AUTOMATA

CAs obtained through conditional derivatives as shown in the previous section are nondeterministic. As one of the main contributions of this work, we now propose an approach for determinizing them into a form that can be used efficiently for regex matching. The approach from which we start and to which we contrast our new method is the naive determinization of CAs to DFAs: The given CA is first converted to its underlying NFA, by making the counter memories an explicit part of control states. The NFA is in turn determinized by the textbook subset construction.

Already the first step, the construction of the NFA, oftentimes explodes since it sacrifices the succinctness of symbolic counters (it is linear to the counter bounds). This initial blow-up is then much amplified in the subset construction, which is exponential to the size of the NFA and hence also to the counter bounds (as, e.g., in the case of the regex `.*a.{`k`}` with its CA in Fig. 1a).

Our answer to this problem is a direct determinization of the CA into a novel type of automata, which we call *counting-set automata (CsAs)*. Control states of counting-set automata produced by our determinization are essentially the states of the corresponding DFA but with the counter memories removed. In order to be able to simulate a run of the DFA, they are equipped with special registers that can hold *sets* of integers, and they use them to compute the right counter memories at runtime. This completely avoids the state space explosion of the naive construction caused by wiring counter memories into control states. Moreover, the simulation is fast because all the manipulations with a counting set can be done in constant time.

6.1 Counting-Set Automata

We now formalize the idea of counting-set automata outlined above. We use the notion of a combined Boolean algebra $\mathbb{I}\times\mathbb{S}$, which allows us to manipulate pairs of predicates from the input algebra $\mathbb{I}$ and the counting-set algebra $\mathbb{S}$. For the purposes of this paper, we assume that predicates in $\Psi_{\mathbb{I}\times\mathbb{S}}$ have the form $\alpha \wedge \beta$ where $\alpha \in \Psi_{\mathbb{I}}$ and $\beta \in \Psi_{\mathbb{S}}$. The conjuction $(\alpha \wedge \beta) \wedge_{\mathbb{I}\times\mathbb{S}} (\alpha' \wedge \beta')$ has the usual meaning of $(\alpha \wedge_{\mathbb{I}} \alpha') \wedge (\beta \wedge_{\mathbb{S}} \beta')$ and $\alpha \wedge \beta$ is satisfiable if both α and β are satisfiable in their respective algebras.

Counting sets. We consider a set-based interpretation of counters where the value of a counter c is a *finite set* rather than a single value. A counter under such an interpretation is referred to as a *counting set*. A *(counting-)set memory* for C is a function $\mathfrak{s} : C \to \mathcal{P}_{\text{fin}}(\mathbb{N})$ such that, for all $c \in C$, $Max(\mathfrak{s}(c)) \leq \boldsymbol{max}_c$.[3] Observe that the set of all set memories for C is *finite*. Counting-set predicates over C form an effective Boolean algebra $\mathbb{S}_C$ called the *counting-set algebra over* C, also denoted just $\mathbb{S}$ when C is clear from the context, whose domain $\mathfrak{D}_{\mathbb{S}}$ is the set of all set memories for C. The set of predicates $\Psi_{\mathbb{S}}$ is the Boolean closure of the basic predicates canIncr_c and canExit_c, hence syntactically the same as in the counter algebra $\mathbb{C}$, but with a different semantics under $\mathbb{S}$:

$$\mathfrak{s} \models \text{canExit}_c \iff Max(\mathfrak{s}(c)) \geq \boldsymbol{min}_c \quad \text{and} \quad \mathfrak{s} \models \text{canIncr}_c \iff Min(\mathfrak{s}(c)) < \boldsymbol{max}_c$$

where $Min(\cdot)$ and $Max(\cdot)$ are the set minimum and maximum, respectively. Intuitively, the conditions test existence of a set element satisfying the same counter condition.

Counting-set automata. A *counting-set automaton* (CsA) is a tuple $A = (\mathbb{I}, C, Q, q_0, F, \Delta)$ where: $\mathbb{I}$ is an effective Boolean algebra called the *input algebra*. C is a finite set of *counters* associated with the counting-set algebra $\mathbb{S}$. Q is a finite set of *states* with $q_0 \in Q$ being the *initial state*. $F : Q \to \Psi_{\mathbb{S}}$ is the *final-state condition*. $\Delta \subseteq Q \times \Psi_{\mathbb{I}\times\mathbb{S}} \times (C \to \mathcal{P}(\boldsymbol{O})) \times Q$ is a finite set of *transitions*. The second component is its *guard*. The third component is the *counting-set operator* in which $\boldsymbol{O} = \{\text{Incr}, \text{Noop}, \text{Rst}, \text{Rst1}\}$ is the set of *counting-set operations*. They are essentially counter operations lifted to sets (note the use of the larger initial letters to distinguish them from the

[3] We write $\mathcal{P}_{\text{fin}}(X)$ for the powerset of X restricted to finite nonempty sets.

counter operations). We also use the different names Rst and Rst1 for the lifting of exit and exit1 to stress their different usage (not only for exiting a loop but also for initialisation when entering the loop as will become clear in Eq. (7)). Sets of counting-set operations assigned to every counter by the counting-set operator are called *combined (counting-set) operations*.

The CsA A is *deterministic* iff the following holds for every two transitions $p\text{-}(\psi_1,f_1)\to q_1$ and $p\text{-}(\psi_2,f_2)\to q_2$ in Δ: if $\psi_1 \wedge \psi_2$ is satisfiable, then $f_1 = f_2$ and $q_1 = q_2$.

Semantics of CsAs. The semantics of an indexed counting-set operation $\textsc{op}_c \in \mathcal{O}$ is the set transformer $upd(\textsc{op}_c)$ defined as follows:

$$upd(\textsc{Incr}_c) = \boldsymbol{\lambda} S.\{n+1 \mid n \in S \wedge n < \boldsymbol{max}_c\} \qquad upd(\textsc{Rst}_c) = \boldsymbol{\lambda} S.\{0\}$$
$$upd(\textsc{Noop}_c) = \boldsymbol{\lambda} S.S \qquad upd(\textsc{Rst1}_c) = \boldsymbol{\lambda} S.\{1\}$$

Then, the counting-set operator $f : C \to \mathcal{P}(\mathcal{O})$ is assigned the counting-set-memory transformer $\mathrm{f} : \mathfrak{D}_{\mathbb{S}} \to \mathfrak{D}_{\mathbb{S}}$ defined as follows:

$$\mathrm{f}(\mathfrak{s}) \stackrel{\text{def}}{=} \boldsymbol{\lambda} c. \begin{cases} \bigcup_{\textsc{op} \in f(c)} upd(\textsc{op}_c)(\mathfrak{s}(c)) & \text{if } f(c) \neq \emptyset \\ \{0\} & \text{if } f(c) = \emptyset \end{cases}$$

That is, (1) if $f(c) \neq \emptyset$, then the value $\mathfrak{s}(c)$ of each counting set c is transformed into the union of the counting sets that result from applying the operations from $f(c)$ on $\mathfrak{s}(c)$, and (2) if $f(c) = \emptyset$, then c is implicitly reset to $\{0\}$ (an implicit Rst). Our determinization procedure creates such transitions when the value of c is irrelevant (when c is a dead variable).

Note that, unlike counter operators of a CA, a counting-set operator f does not induce any guard. The guard is rather a separate component of the transition. This is because CsA transitions produced in the CA-to-CsA determinization need guards that are partially independent of the operations of f. In particular, we will we need to distinguish cases such as $\neg\textsc{canExit}_c \wedge \textsc{canIncr}_c$, $\textsc{canExit}_c \wedge \neg\textsc{canIncr}_c$, or $\textsc{canExit}_c \wedge \textsc{canIncr}_c$. The guard hence cannot be induced by f alone.

Note also that, unlike in CAs, the updates are defined for *indexed* operations. The reason is that the semantics of the Incr operation is restricted to never produce values greater than $\boldsymbol{max}_c$.

Finally, the *language of the CsA* A is defined through its underlying *configuration FA*, $FA(A)$, as $\mathcal{L}(A) := \mathcal{L}(FA(A))$. The states of $FA(A)$ are *configurations* of A, namely, tuples of the form $(q,\mathfrak{s}) \in Q \times \mathfrak{D}_{\mathbb{S}}$ consisting of a state q and a counting-set memory $\mathfrak{s}$. There are finitely many such configurations. The initial state of $FA(A)$ is the *initial configuration* $(q_0, \{c \mapsto \{0\}\}_{c\in C})$ of A. A transition $\tau = p\text{-}(\alpha\wedge\beta,f)\to q \in \Delta$ is *enabled* in a configuration $(p,\mathfrak{s})$ iff α is satisfiable and $\mathfrak{s} \in [\![\beta]\!]_{\mathbb{S}}$, meaning that $\mathfrak{s}$ satisfies the counter guard β. If τ is enabled in $(p,\mathfrak{s})$, then $FA(A)$ contains the transition $(p,\mathfrak{s})\text{-}(\alpha)\to(q,\mathrm{f}(\mathfrak{s}))$. Finally, a state $(q,\mathfrak{s})$ of $FA(A)$ is *final* iff $\mathfrak{s} \models F(q)$.

Example 6.1. An example of a CsA is in Fig. 1b. It uses intuitive notations that were also introduced in Section 2 as abbreviations for the operations of the counting-set data structure. Counting-set operators are depicted as assignments to c, Rst is represented as $\{0\}$ on the right of the assignment, Rst1 is represented by $\{1\}$, Incr by $c+1$, and Noop by c. Multiple transitions between the same states and with the same updates are merged into one with a simplified guard. An example whose notation closely follows the formal development is in Fig. 3. □

Runtime efficiency of counting sets. A major reason for choosing CsAs as the target kind of machine for determinization of CAs is that pattern matching with CsAs is fast. Using the data structure explained in Section 2, all the basic counting-set tests and updates, namely, $\textsc{canIncr}_c$, $\textsc{canExit}_c$, Noop, Incr, Rst, and Rst1, can be implemented to run in constant time regardless of the size of the counting set and the value $\boldsymbol{max}_c$ (assuming constant-time complexity of integer arithmetic operations). Moreover, almost all combined counting-set operations can be implemented to run in constant time too. In particular, when at most one counting-set operation of a given

combined operation returns a set other than $\{0\}$ or $\{1\}$, their union can be computed in constant time. If this is not the case, the union is linear to the size of the sets computed by the particular counting-set operations (which is at most $\boldsymbol{max}_c$). The only operations that may return sets other than $\{0\}$ or $\{1\}$ are Noop and Incr. We denote a transition whose counting-set operator f assigns to some counter c the result of a combined operation $f(c)$ that contains both Noop and Incr as *slow*. A CsA that has slow transitions is called *slow*, and a CsA that does not have them is called *fast*. Slow CsAs are fortunately rare in practice (cf. Section 7).

When a fast CsA is used in pattern matching, tests and updates of one counting set then take $O(1)$ time, which in turn gives $O(|C|)$ for all counting sets and their unions. *This is our major achievement: the independence of the running time from the counter bounds.*

6.2 Encoding DFA Powerstates as CsA Configurations

In order to build intuition needed for understanding our determinization algorithm, we will first concretize how the configurations of a CsA can encode states of a DFA corresponding to the NFA $FA(A)$ underlying a given CA $A = (\mathbb{I}, C, Q, q_0, F, \Delta)$. First, recall that, since A is converted into $FA(A)$ by making the counter memories explicit parts of control states, the states of $FA(A)$ are pairs $(p, \mathfrak{m})$ consisting of a state p of A and a counter memory $\mathfrak{m}$. Second, assume that $FA(A)$ is determinized using the textbook subset construction.[4] We denote the result as $DFA(A)$ from now on. Then, the states of $DFA(A)$ are sets of states of $FA(A)$, i.e., sets of pairs $(p, \mathfrak{m})$, which we will call *powerstates*. The control states of the CsA A' built by our CA-to-CsA determinization will be subsets of the set Q of states of the CA A. The configurations of A' will thus be pairs $(R, \mathfrak{s})$ where $R \subseteq Q$ is a CsA control state, i.e., a set of states of A, and $\mathfrak{s} : C \to \mathcal{P}_{\text{fin}}(\mathbb{N})$ is a counting-set memory. Let us now consider how $\mathfrak{s}$ can be interpreted in this context.

Naive encoding. A naive interpretation of a CsA configuration $(R, \mathfrak{s})$ is a DFA state containing all pairs $(r, \mathfrak{m})$ such that $r \in R$ and, for all $c \in C$, $\mathfrak{m}(c)$ can be any value from $\mathfrak{s}(c)$. The set of the counter memories $\mathfrak{m}$ is then isomorphic to the Cartesian product $\prod_{c \in C} \mathfrak{s}(c)$ of the sets $\mathfrak{s}(c)$ assigned to the counters, and the entire powerstate is the Cartesian product $R \times \mathfrak{m}$ of the set of states and the set of counter memories. The naive interpretation, however, is too impractical as it cannot express any dependence of a counter memory on the CA state (every state can be paired with each considered memory) nor any mutual dependence of values of different counters within a counter memory (every possible value of a counter c can be paired with every possible value of any other counter d). Most DFAs compiled from real-life regexes do not fit into this representation. For instance, the DFA configuration $\{(q, c = 0), (s, c = 0), (s, c = 1)\}$ of the CA from Fig. 1 in Section 2 could not be represented by a CsA configuration because q and s appear with different sets of counter values.

Encoding with counter scopes. Our key observation how to resolve the above problem (at least for many real-life scenarios) is to take advantage of that not every counter is "used" at every CA state. In fact, the value of a counter is usually implicitly 0 at most states except a few. If these states are known, the implicit zeros do not have to be remembered explicitly in the counting sets, and the encoding becomes much more flexible. To formalize this, we introduce the notion of the *scope of a*

[4] The DFA produced by the textbook subset construction from a *simple* FA $\mathcal{A} = (\mathbb{I}, Q, q_0, F, \Delta)$ will have $\mathcal{P}(Q)$ as the set of states, transitions $S\text{-}(\alpha)\!\!\rightarrow\{r \in Q \mid s\text{-}(\alpha)\!\!\rightarrow r \in \Delta, s \in S\}$, the initial state $\{q_0\}$, and as the final states all those intersecting F. We note that to determinize a CA which is not simple, one could start from the more sophisticated version of the subset construction for symbolic automata of [?], which avoids explicit generation of all minterms.

counter that over-approximates the set of states where a counter c can have a non-zero value and that is easy to compute.[5] The scope is defined inductively as the smallest set of states $\sigma(c)$ such that

(1) $q \in \sigma(c)$ if there is a transition to q with either $\textsc{incr}_c$ or $\textsc{exit1}_c$, or
(2) there is a transition to q from a state in $\sigma(c)$ with the $\textsc{noop}_c$ operation.

In other words, the scope of c spreads from an increment of c along the transition relation until a transition with $\textsc{exit}_c$.

The DFA *powerstate encoded by a CsA configuration* $(R, \mathfrak{s})$ can then be formally defined as the set $(R, \mathfrak{s})^{DFA}$ of configurations $(r, \mathfrak{m})$ of the CA A such that $r \in R$ and, for all $c \in C$, $\mathfrak{m}(c) \in \mathfrak{s}(c)$ if $c \in \sigma(r)$, else $\mathfrak{m}(c) = 0$. We call the powerstates of $DFA(A)$ that can be encoded by CsA configurations *Cartesian*, and call the entire DFA Cartesian if all its powerstates are Cartesian.

Example 6.2. The powerstates of the $DFA(A)$ of the CA A from Fig. 1a are indeed Cartesian (as discussed in Section 2) because q_0 is not in the scope of c. The encoding of powerstates by CsA configurations is also illustrated in Section 2 and later also in Example 6.4. □

The Cartesian encoding still cannot express all kinds of DFA powerstates. In particular, it cannot express more subtle dependencies of counter values on the state, and dependencies of counter values of different counters on each other, which mainly concerns CAs with nested counting loops compiled from regexes with nested counting sub-expressions. Example 6.5 discusses a regex that leads to a non-Cartesian CA. However, we later present a strong empirical evidence that a significant majority of real-life regexes lead to Cartesian CA.

6.3 Generalized Subset Construction

We will now describe the core of our CA-to-CsA determinization. It is built on top of the textbook subset construction for NFAs. We use the CA from Fig. 3a as a running example through the section. We make a simplifying assumption that the input CAs are simple (different character classes on their transitions do not overlap). This is satisfied by CAs generated by the derivative construction from Section 5 since their transitions are labeled by minterms of the original regex. The assumption could be dropped and the construction could be relatively easily generalized in the style of symbolic automata determinization of [**?**].

Let $A = (\mathbb{I}, C, Q, q_0, F, \Delta)$ be a simple CA with the scope function $\sigma : Q \to \mathcal{P}(C)$. The algorithm produces the deterministic CsA $A' = (\mathbb{I}, C, Q', S_0, F', \Delta')$ whose components are constructed as described below. Namely, control states of A', called powerstates, are subsets of Q, i.e., $Q' \subseteq \mathcal{P}(Q)$. The initial powerstate is $S_0 = \{q_0\}$. A powerstate $S \in Q'$ is final iff the final condition holds for some of its elements, i.e., $F'(S) \stackrel{\text{def}}{=} \bigvee_{q \in S} F(q)$. The sets Δ' and Q' are constructed by a fixpoint computation that explores the state space reachable from S_0. During the construction, transitions starting from previously reached powerstates are constructed and included together with their target states into Δ' and Q', respectively, until no new powerstates can be reached.

Transitions starting from a given control state R of the CsA A' are constructed to update the runtime values of counting sets such that they simulate transitions of the DFA corresponding to the CA A. Assume a CsA configuration $(R, \mathfrak{s})$ and a DFA transition $(R, \mathfrak{s})^{DFA} \text{-}(\alpha)\!\!\rightarrow P$ from the DFA powerstate encoded by $(R, \mathfrak{s})$ over an input minterm α. The simulating CsA transition must transform $(R, \mathfrak{s})$ into $(R', \mathfrak{s}')$ with $(R', \mathfrak{s}')^{DFA} = P$. The simulated DFA transition was constructed from α-transitions of the NFA $FA(A)$ that are actually instantiations of the CA α-transitions enabled

[5] Computing the precise set of states where a counter c can have a non-zero value would require a reachability analysis in the general case (since some of the transitions may never be executable—think of simultaneously counting with counters c and d such that $\textsc{canIncr}_c < \textsc{canExit}_d$, then the exit transition for d will never be taken). For the CAs made by our derivative construction, the scope, however, corresponds to this set precisely—no transitions that are never executable are generated.

in configurations $(r, \mathfrak{m}) \in (R, \mathfrak{s})^{DFA}$. The simulating CsA transition will be constructed from these CA transitions. They can be identified by (1) their source state, which must be in R, (2) an alphabet minterm $\alpha \in \Sigma$ where Σ is the set of minterms over all input predicates in the CA A, and (3) their compatibility with a particular set of enabled/disabled counter guards. This set of guards belongs to the set of minterms $\Gamma_{R,\alpha}$ of the set of counter guards on the α-transitions originating in R:

$$\Gamma_{R,\alpha} \stackrel{\text{def}}{=} \mathit{Minterms}(\{grd(\mathrm{OP}_c) \mid r \text{ -}(\alpha,f)\!\!\rightarrow s \in \Delta, r \in R \wedge c \in \sigma(r), \mathrm{OP}_c \in f\}).$$

Hence, the CsA will have a transition leaving R for each $\alpha \in \Sigma$ and $\beta \in \Gamma_{R,\alpha}$, and the transition will be built from the set of CA α-transitions originating in R and *consistent* with β:

$$\Delta_{R,\alpha,\beta} \stackrel{\text{def}}{=} \{r \text{ -}(\alpha,f)\!\!\rightarrow s \in \Delta \mid r \in R, \mathbf{\mathit{Sat}}(\varphi_f \wedge \beta)\}.$$

Its target is the set T of all target states of the transitions in $\Delta_{R,\alpha,\beta}$, and its guard is $\alpha \wedge \beta$.[6]

The remaining component is the counting-set operator f'. It must summarize the updates of the counter values on transitions of $\Delta_{R,\alpha,\beta}$ as updates of the respective counting sets. The values of counters that are out of scope, hence implicitly zero, will not be tracked in counting sets. Tracking the value of a counter hence starts when A' simulates a transition of A entering the scope of the counter, and ends when no state from the scope is present in the target CsA state.

Let $\Delta_{R,\alpha,\beta}(c)$ be the set of transitions in $\Delta_{R,\alpha,\beta}$ with the target state in the scope of c. The counting-set operator f' is built in the form $f'(c) \stackrel{\text{def}}{=} \{op(\tau, c) \mid \tau \in \Delta_{R,\alpha,\beta}(c)\}$. Here, $op(\tau, c)$ denotes the counting-set operation that, given a CA transition $\tau = p\text{-}(\alpha,f)\!\!\rightarrow q$, transforms the set of possible values of the counter c at the state p to the set of values obtained at q after taking the transition. It is defined in Eq. (7) on the right. The set operation induced by the CA transition corresponds to the counter operation on the transition. In the third and fourth case, when the CA transition comes from out of the scope, it is certain that the counter can only have the value 0, which is the same value as produced by EXIT (or EXIT1 when the counter is immediately incremented). The resulting CsA transition is therefore $S\text{-}(\alpha\wedge\beta,f')\!\!\rightarrow T$. Note that $f'(c)$ ends up empty when the target powerstate is fully out of the scope of c, which semantically corresponds to the implicit reset to $\{0\}$.

$$op(\, p\text{-}(\alpha,f)\!\!\rightarrow q\,, c) \stackrel{\text{def}}{=} \begin{cases} \textsc{Noop} & \text{if } f(c) = \textsc{noop} \wedge p \in \sigma(c) \\ \textsc{Incr} & \text{if } f(c) = \textsc{incr} \wedge p \in \sigma(c) \\ \textsc{Rst} & \text{if } f(c) = \textsc{noop} \wedge p \notin \sigma(c) \\ \textsc{Rst1} & \text{if } f(c) = \textsc{incr} \wedge p \notin \sigma(c) \\ \textsc{Rst} & \text{if } f(c) = \textsc{exit} \\ \textsc{Rst1} & \text{if } f(c) = \textsc{exit1} \end{cases} \tag{7}$$

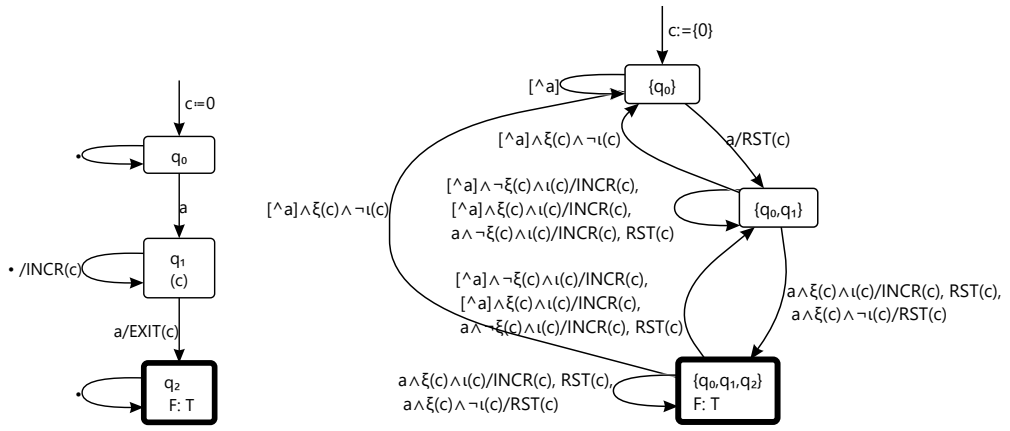
Observe that A' is deterministic since, for any two distinct transitions $S\text{-}(\alpha_1,f_1)\!\!\rightarrow S_1$ and $S\text{-}(\alpha_2,f_2)\!\!\rightarrow S_2$, the condition $\alpha_1 \wedge \alpha_2$ is unsatisfiable by virtue of minterms.

THEOREM 6.3. *For the CA A and the CsA A' above, we have $\mathcal{L}(A') \supseteq \mathcal{L}(A)$ and $|Q'| \leq 2^{|Q|}$.*

PROOF (IDEA). The language inclusion is proved by showing that the configuration automaton $FA(A')$ of A' simulates $DFA(A)$, more concretely, that each configuration $(R, \mathfrak{s})$ of A', a state of $FA(A')$, simulates the powerstate $(R, \mathfrak{s})$ of $DFA(A)$. The bound on the size of the state space follows from that states of the CsA are sets of states of the CA. □

Example 6.4. Consider the CA in Fig. 3a that has states q_0, q_1, and q_2. The state q_0 is initial, the final condition of q_2 is $\top$, and it is $\bot$ for q_0 and q_1. The set of counters is $C = \{c\}$ with $\sigma(c) = \{q_1\}$ (i.e., c is not used and hence implicitly 0 in q_0 and q_2). Finally, $\Sigma = \{\mathtt{a}, \mathtt{[\^{}a]}\}$. In Fig. 3a, we compactly represent transitions over all minterms from Σ using $\bullet$. The determinization starts exploring the CsA from its initial state $S_0 = \{q_0\}$.

[6] Recall that the predicates in $\Psi_{\mathbb{C}}$ and $\Psi_{\mathbb{S}}$ are syntactically the same.



(a) The CA for •*a•{4,8}a (b) Determinization of the CA into a CsA

Fig. 3. From a regex via a CA to a deterministic CsA. We are using a notation closely following the formal development. We only use $\text{OP}(c)$ instead of OP_c and abbreviate CANEXIT_c by $\xi(c)$ and CANINCR_c by $\iota(c)$.

Let us focus on the transitions for the input minterm $\alpha = \mathsf{a}$. Two transitions leaving q_0, namely $\delta_1 = q_0 \text{-}(\mathsf{a},\text{NOOP}_c)\!\!\rightarrow\! q_0$ and $\delta_2 = q_0 \text{-}(\mathsf{a},\text{NOOP}_c)\!\!\rightarrow\! q_1$, both with no guard on c, hence $\Gamma_{S_0,\alpha} = \{\top\}$. The guard $\top$ is thus the only choice for the counter minterm β. The set $\Delta_{R,\alpha,\beta}$ of transitions consistent with α and β then contains both a-transitions δ_1 and δ_2 originating from q_0. Since δ_2 is entering the scope of c, it generates the counting-set operation RST_c according to the third case of Eq. (7). Since δ_1 stays out of the scope, it does not generate any counting-set operations. We obtain the counting-set operator $f' = \{\text{RST}_c\}$ and generate the CsA transition $\tau_1 = \{q_0\} \text{-}(\mathsf{a}\wedge\beta,\{\text{RST}_c\})\!\!\rightarrow\! \{q_0, q_1\}$.

Next, let us focus on the a-transitions from $S_1 = \{q_0, q_1\}$. Here, $\Gamma_{S_1,\mathsf{a}}$ has the following three satisfiable elements: $\text{CANEXIT}_c \wedge \text{CANINCR}_c$, $\neg\text{CANEXIT}_c \wedge \text{CANINCR}_c$, and $\text{CANEXIT}_c \wedge \neg\text{CANINCR}_c$ (the guard $\neg\text{CANEXIT}_c \wedge \neg\text{CANINCR}_c$ is excluded as it is never satisfied for non-empty sets of positive integers). Let us generate a transition for the second case, $\beta = \neg\text{CANEXIT}_c \wedge \text{CANINCR}_c$. We obtain $\Delta_{S_1,\mathsf{a},\beta} = \{q_0 \text{-}(\mathsf{a},\text{NOOP}_c)\!\!\rightarrow\! q_0,\ q_0 \text{-}(\mathsf{a},\text{NOOP}_c)\!\!\rightarrow\! q_1,\ q_1 \text{-}(\mathsf{a},\text{INCR}_c)\!\!\rightarrow\! q_1\}$. As before, the first transition does not contribute to f' as it stays out of the scope, and the second transition adds RST_c. The third transition adds INCR_c (the second case of Eq. (7)). The resulting CsA transition is thus $\tau_2 = S_1 \text{-}(\mathsf{a}\wedge\neg\text{CANEXIT}_c\wedge\text{CANINCR}_c,\{\text{INCR}_c,\text{RST}_c\})\!\!\rightarrow\! S_1$. The rest of the construction is analogous.

Last, let us also illustrate the simulation of $DFA(A)$ by the constructed CsA transitions. On the word aa, the DFA would execute the run $\{(q_0, c = 0)\} \text{-}(\mathsf{a})\!\!\rightarrow\! \{(q_0, c = 0), (q_1, c = 0)\} \text{-}(\mathsf{a})\!\!\rightarrow\! \{(q_0, c = 0), (q_1, c = 0), (q_1, c = 1)\}$. The simulating run of our CsA would start in the initial configuration $\{\{q_0\}, c \in \{0\}\}$. The transition τ_1 would produce the configuration $\{\{q_0, q_1\}, c \in \{0\}\}$ (since $\text{RST}(\{0\}) = \{0\}$) from where τ_2 would produce $\{\{q_0, q_1\}, c \in \{0, 1\}\}$ (since $\text{INCR}(\{0\}) = \{1\}$ and $\text{RST}(\{0\}) = \{0\}$). The sequence of configurations precisely encodes the sequence of the DFA powerstates, that is, the sequnce $(\{\{q_0\}, c \in \{0\}\})^{DFA} = \{(q_0, c = 0)\}$; $(\{\{q_0, q_1\}, c \in \{0\}\})^{DFA} = \{(q_0, c = 0), (q_1, c = 0)\}$; and $(\{\{q_0, q_1\}, c \in \{0, 1\}\})^{DFA} = \{(q_0, c = 0), (q_1, c = 0), (q_1, c = 1)\}$ (recall that q_0 is not in the scope of c hence c has implicitly the value 0 there). □

6.4 Uniformity: A Sufficient Semantic Correctness Criterion

Given a CA A, we produce a CsA A' that may overapproximate A in terms of the language. We explain how this may happen and present conditions under which the language stays unchanged. In particular, the overapproximation is caused by non-Cartesian powerstates of $DFA(A)$. (Recall that, in a Cartesian powerstate, states in the scope of a counter must appear with the same set

of values of that counter.) A configuration of the CsA cannot encode a non-Cartesian powerstate precisely, it can only overapproximate it. A larger powerstate may then accept a larger language.

Example 6.5. Take $R = (\mathtt{a}|\mathtt{aa})\{5\}$ and the CA(R) shown in Fig. 4. After reading the word *aa*, *DFA*(CA(R)) reaches the powerstate $\{(q_0, c = 1), (q_0, c = 2), (q_1, c = 2)\}$, which is not Cartesian because both states are in the scope of the counter c but are paired with different counter values. Our CsA would reach the configuration $(\{q_0, q_1\}, c \in \{0, 1, 2\})$, which encodes the larger powerstate $\{(q_0, c = 0), (q_0, c = 1), (q_0, c = 2), (q_1, c = 0), (q_1, c = 1), (q_1, c = 2)\}$ where both states appear with both counter values. □

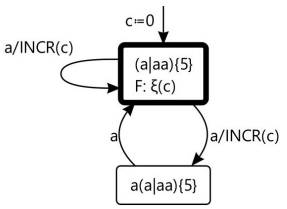


Fig. 4. $CA((\mathtt{a}|\mathtt{aa})\{5\})$.

Uniformity. We now introduce the so-called *uniformity* of a CA as a property under which determinization preserves the language. Uniformity prevents creation of non-Cartesian powerstates. It includes two conditions.

The first condition prevents the kind of scenario from Example 6.5. For each DFA transition τ', it requires that every CA state q that is in the scope of some counter c within the DFA state to which τ' leads receives the same set of values of c. This requires testing whether the sets of transitions covered by τ' and incoming to every such CA state q induce the same CsA operations for c.

The second condition prohibits two counters from being active at once, a scenario which arises from regexes with nested counting. Indeed, the relation between values of two simultaneously active counters may easily become more intricate than what can be expressed by a Cartesian product of two sets (consider, e.g., the regex $a?(a\{1\}a)\{2\}$ and the word *aaa*). The condition requires testing that no state appears in the scope of two counters.

Formally, given a CsA transition $\tau' = S\text{-}(\alpha \wedge \beta, f')\!\rightarrow\! T$, a counter c, and a CA state $q \in \sigma(c)$, we define the set $f'_q(c)$ of *incoming CsA operations* for c induced by the incoming transitions of q from which τ' is built (α-transitions consistent with β originating in S) as follows:

$$f'_q(c) \stackrel{\text{def}}{=} \{op(\tau, c) \mid \tau \in \Delta_{S,\alpha,\beta(c)} \wedge \text{the target of } \tau \text{ is } q\} .$$

We call the transition τ' *uniform* iff, for each counter $c \in C$, any two states $q, r \in \sigma(c) \cap T$ have the same sets of incoming CsA operations, i.e., $f'_q(c) = f'_r(c)$. The CA A is then *uniform* if all transitions of A' are uniform and if no state of A appears in the scope of two counters.

Theorem 6.6. *If a CA A is uniform, then $\mathcal{L}(A) = \mathcal{L}(A')$.*

Proof (idea). By showing bisimilarity between states q of $FA(A')$, i.e., configurations of the CsA A' and powerstates q^{DFA} of $DFA(A)$. □

Uniformity can be checked on the fly, while constructing A'. It is also automatically implied when the CA is constructed from certain classes of regexes, as discussed below.

6.5 Syntactic Correctness Criteria

Uniformity is only a semantic property. Below, we show examples of actual regexes that do and do not lead to uniform CAs and discuss some simple syntactic classes of regexes that imply uniformity. A detailed study of syntactic classes of regexes that guarantee uniformity is, however, beyond the scope of this paper and a part of our future work.

The regexes that induce non-uniform CAs are often those where, intuitively, there is a position in some input text that may either be matched against the first character of a counted sub-expression or against some inner character of the same sub-expression. In such a situation, there may be two runs of the induced CA: one that increments the associated counter (the increment happens) at

that position and moves to some state q, and the other that leaves the counter as it is, while in its scope, and moves into a different state r. The counter value then depends on the state: it is different in q and in r. The corresponding DFA state is then non-Cartesian and the CA is non-uniform.

Example 6.7. We present several commented examples of regexes with non-uniform CAs where our determinization overapproximates the language of the obtained CsA.

- `(a|ab|ba){5}` — the string *aba* could be matched as "a" followed by "ba", having incremented the counter twice, or as "ab" that is followed by the prefix "a" of "ab", having incremented the counter once only.
- `a{1,3}a{3}` — this case can be explained similarly as the previous one. Alternatively, note that, assuming that our translation to a CA produces two counters, say c_1 and c_2, then after reading n letters a, the CA needs to remember that $c_1 + c_2 = n$. Such non-trivial relations between counter values are not Cartesian.
- `.*(aa){6}` — assuming a sequence of a's on the input, the counter may be either incremented on odd characters and left unchanged on even ones, or the other way around. As the counter values depend on the position within the "aa" (and hence on the CA state), the CA cannot be uniform. Note that the prefix `.*` is quite usual as it corresponds to searching for the regex `(aa){6}` anywhere in the input string.
- `.*(a{2}){2}` — after reading *aa*, if the value of the outer counter is 1, then the value of the inner counter must be 0. This is a non-trivial relation between the values of the two counters, which is not Cartesian. Nested counting is often problematic, however, many of such examples may still be solved quite efficiently by unfolding one of the counters. □

Syntactic classes of regexes that guarantee uniformity. A simple class of regexes that guarantees uniformity is a generalization of the class of *monadic* regexes of [?] (where counting is allowed over character classes only). Namely, it includes regexes with counting loops of the form

$$(\alpha_1 \dots \alpha_n)\{\ell, k\} \text{ s.t. } [\![\alpha_1]\!] \text{ is disjoint from every } [\![\alpha_i]\!], 1 < i \leq n.$$

Intuitively, the disjointness with α_1 ensures that the generated CA will only be able to process α_1 through an increment transition at the beginning of a new iteration of the loop, with no possibility of having a conflicting NOOP transition that could read the same symbol inside the body of the loop (which is exactly what happens with the second symbol a in Example 6.5). The CsA compiled form this class are also guaranteed to be fast.

7 EXPERIMENTAL EVALUATION

We have implemented our approach in a C# prototype called `CA` available at [?] (see [?] for details how to efficiently implement CsAs) and evaluated its pattern matching capabilities against other state-of-the-art regex matchers on patterns that use the counting operator. We focused on comparison against Google's `RE2` library [?][7], an automata-based matcher designed to be fast, predictable, and resilient against ReDoS attacks. We also include other three efficient matchers into the comparison, namely the standard GNU `grep` program [?] (version 3.3), the `.NET` standard library regex matcher from `System.Text.RegularExpressions` [?], and Symbolic Regex Matcher (`SRM`) [?].

Let us shortly summarize how the tools work. The main algorithms of `RE2` and `grep` implement optimized versions of the Thompson's on-the-fly determinization where the constructed DFA states are cached. The construction has a bound on the size of the DFA—if the bound is reached, the so-far constructed DFA states are flushed to avoid consuming too much memory. In some situations

[7] We used the version 2019-01-01 of `RE2` via the command line interface `re2g` from https://github.com/akamai/re2g.

when caching is found ineffectual, RE2 turns the caching off, and the performance can drop even lower (see the description in [?] for details). We note that RE2 rejects an input regex if it contains a counting operator with a bound bigger than 1,000. SRM is based on *symbolic derivatives* constructed on the fly, also in the spirit of the Thompson's algorithm, and, likewise, bases its efficiency on caching (in fact, SRM is quite close to an implementation of the Thompson's algorithm over CAs with caching). The .NET matcher uses a backtracking algorithm over NFAs, while our CA eagerly constructs a deterministic CsA for the input regex. The former four are mature tools, and especially RE2 and grep contain many high- and low-level optimizations, such as using the Boyer-Moore algorithm [?] to skip over many characters that are known to not be a part of a match. RE2 and grep are compiled programs while CA, SRM, and .NET run within the .NET Framework (therefore, they have some inherent overhead due to the *just-in-time* compilation at start-up and its inability to use advanced code optimizations, as well as garbage collection). Note that even though the tools based on the on-the-fly subset construction (RE2, grep, and SRM) are linear to the lenght of the text, they still take space exponential to the counter bounds in the worst case, by creating sets of the size linear to the counter bounds, exponential to their decadic encoding used in the regex.

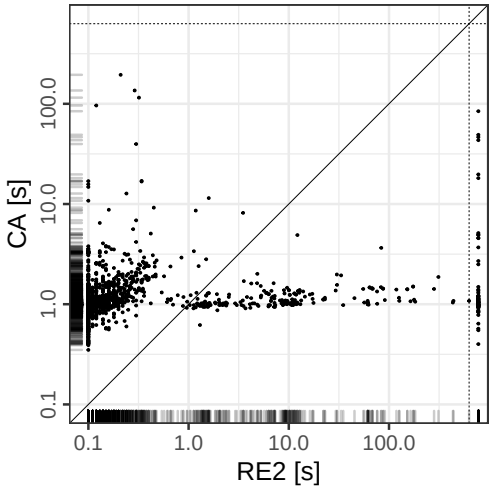
We run our benchmarks on a machine with the Intel(R) Xeon(R) CPU E3-1240 v3 @ 3.40 GHz running Debian GNU/Linux (we use the Mono platform [?] to run .NET tools). To avoid issues with generating exact matches, which might differ for different tools, the tools were run in the setting where they counted the number of lines matching[8] the given regex (e.g. the -c flag of grep).

7.1 ReDoS Resiliency

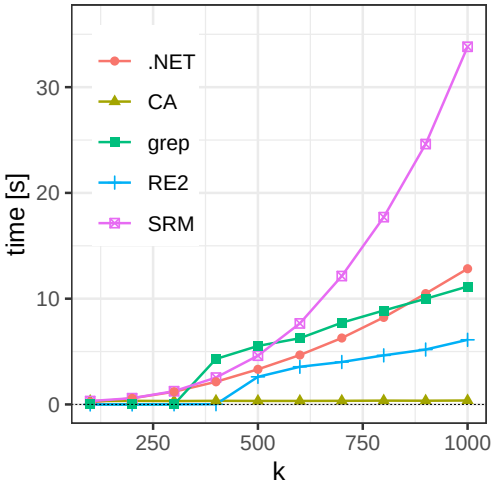
Our main experiment focuses on the resilience of the matching engines against ReDoS attacks. The regexes used for this experiment were selected (1) from the database of over 500,000 real-world regexes coming from an Internet-wide analysis of regexes collected from over 190,000 software projects [?]; (2) from databases of regexes used by *network intrusion detection systems* (NIDSes), in particular, Snort [?], Bro [?], Sagan [?], and the academic papers [??]; (4) the RegExLib database of regexes [?]; and (5) industrial regexes from [?], used for security purposes. From these, we created our set of benchmarks by the following steps:

(1) We selected regexes that contained counting loops whose sum of upper bounds was larger than 20. This let us focus on regexes where the use of counting makes sense (there are surprisingly many regexes occurring in practice where the use of a counting loop is unnecessary, e.g., regexes containing sub-expressions similar to a{0,1} or even just a{1}). Moreover, we also removed all except 26 regexes with counters bigger than 1,000, which cannot be handled by RE2. We left the 26 regexes as representatives of "large" counters. This left us with 5,000 regexes.
(2) Then, we filtered out regexes R such that either CA(R) was not uniform (i.e., the CsA produced by our algorithm was not precise, cf. Section 6.4), or such that the *CsA* was not fast (i.e. not all counting-set operators were constant-time, cf. Section 6.1). After this step, a vast majority, 4,429 of the regexes, remained.
(3) For the regexes that remained, we used a lightweight ReDoS generator designed to exploit counting (cf. Section 7.3) to generate ~10 MiB long input texts. In particular, we managed to generate "adversarial" input texts for 1,789 regexes (for the rest of the regexes, either the underlying state space was too small, so the generator could not construct the text, or the generation hit the timeout of 600 s). Our benchmark data set is available at [?].

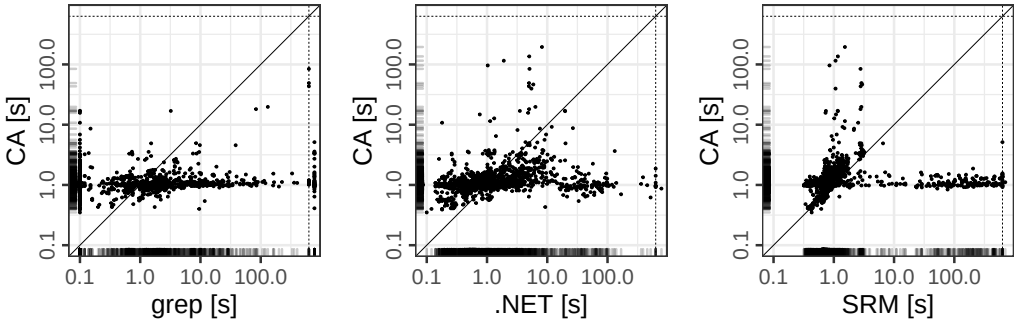
[8]We consider the standard semantics of "matching" used by grep, i.e., a line matches a regex R if it contains a string that is in $\mathcal{L}(R)$, unless it contains start-of-line (^) or end-of-line ($) anchors, in which case the matched string needs to occur at the start and/or at the end of the line respectively.



(a) The comparison of running times of CA and RE2 on our benchmark set (CA wins: 287/1,789)



(b) Running times of the tools on the regex "(_a){k}_a" where k is a parameter



(CA wins: 862/1,425) (CA wins: 708/1,789) (CA wins: 345/1,789)

(c) The comparison of running times of CA with grep, .NET, and SRM on our benchmark set

Fig. 5. Graphs with results of our experiments. Note that, in (a) and (c), the axes are logarithmic, the dashed lines denote the timeout (600 s), and the data points between the dashed lines and the edge of a plot represent benchmarks where the tool did not run successfully. We also provide the number of times CA won.

We ran all tools on the generated benchmarks (counting the number of lines of the input text matching the regex) and give scatter plots comparing the running times of the tools in Fig. 5a and Fig. 5c (the timeout was 600 s). On the bottom and the left-hand side of every plot, there are rug plots illustrating the distribution of the data points. Note that the axes are logarithmic, so the difference between data points grows as these points are away from zero (in particular, differences of values smaller than 1 s are negligible). The semantics of regexes supported by grep differs from the one supported by other tools, so we only considered the cases when the number of matches was the same when comparing with grep). In the plots, the data points between the dashed lines and edges of the plots represent errors, e.g. due to the regex being rejected (for counters >1,000 for RE2) or being interpreted using a different semantics (in the case of grep).

In Fig. 5a, we compare CA with RE2. We wish to point out the following interesting observations. Although RE2 wins more often on the whole benchmark set (our prototype does not include the

Table 1. Statistics for the graphs in Fig. 5 (times are given in seconds). For CA, we provide several times: "total" is the total time, "CA" is the time for translating a regex into a (nondeterministic) CA, "CsA" is the time of determinization of the CA into a CsA, and "match" is the time spent when matching the input text.

	RE2	grep	.NET	SRM	CA			
					total	CA	CsA	match
mean	36.11	34.38	9.12	26.78	1.73	0.05	0.23	0.69
median	0.10	0.70	0.76	0.73	1.03	0.03	0.04	0.68
std. dev	157.05	147.17	52.10	106.16	7.27	0.29	2.73	0.29
timeouts	1	11	8	16	0			

many advanced optimizations present in RE2), there is a number of benchmarks (287) where its performance significantly deteriorates, and CA is faster. In particular, there are 89 benchmarks where the time of RE2 is bigger than 10 s, i.e., its speed drops below 1 MiB/s (we consider this speed of processing denotes a successful ReDoS attack, even though the limit may be significantly larger in practice[9]). For CA, the number of benchmarks that took over 10 s was only 22; in fact, all except 3 benchmarks finished within 100 s—the blow-up in these 3 benchmarks is not caused by the counters but rather by many "|" and "?" operators, so over 70 % of the total time is spent by constructing the CsA. If used, e.g., in an NIDS, the CsA would be created only once and then used for matching giga-/terabytes of data, so the initial overhead could be neglected.

Comparing with the other tools (Fig. 5c) and also clearly visible in the corresponding rug plots and the statistics in Table 1, we can observe that the performance of CA is much more robust than the performance of the other tools; the mean time and standard deviation of CA is significantly lower than the rest of the tools. In particular, from the benchmarks where CA was faster than RE2, the time of CA on all except two benchmarks was almost the same (including them, the standard deviation was 0.37). We provide four times for CA: "total": the total user time of matching (measured using the GNU `time` utility), "CA": the time for translating the input regex into a CA, "CsA": the time it took to determinize the CA into a CsA, and "match": the time of matching the input text with the CsA. Note that, in the tables, there is a noticeable discrepancy between the sum "CA" + "CsA" + "match" and "total", which is due to the .NET Framework overhead, such as just-in-time compilation and (in particular) the garbage collector.

In Table 2, we give a selection of interesting benchmarks. These contain benchmarks that are difficult for usually more than one tool. We emphasize the benchmarks coming from the NIDSes Snort and Bro. Notice that, for most of them, matching using RE2 (and also other tools) gets extremely slow. Slow matching over these regexes can have disastrous consequences for network security, potentially completely eliminating a given NIDS.

The CsAs produced by CA were also much smaller than the corresponding DFAs. The CsAs have on average 29 states (median: 7) and 306 transitions (median: 11). On the other hand, classical NFAs constructed from the regexes have on average 112 states (median: 52), and when determinized, the resulting DFAs have on average 2,802 states (median: 67) and 10,384 transitions (median: 107). Using CsAs significantly lowers the chance that determinization explodes.

7.1.1 The Effect of Nondeterministic Counting. We say that a regex contains *nondeterministic counting* if, when translated into a CA A using the algorithm in Section 5, there is a word w such that A can over w reach two configurations with different values of some counter.

[9]The required processing speed depends on the application. NIDSes performing deep packet inspection may require a line-processing speed of units or tens of GiB/s [?], while application servers validating user inputs may suffice with units or tens of MiB/s.

Table 2. Selection of interesting benchmarks. "TO" denotes a timeout (600 s) and "—" denotes an error. Due to space constraints, in the "Regex" column, "…" denotes omitted parts of the regexes (we tried to preserve the parts containing occurrences of the repetition operator) and "~" denotes breaking a regex into two lines. In the column source, Sw denotes the regexes collected in [?] from software projects.

Source	Regex	RE2	grep	.NET	SRM	CA			
						total	CA	CsA	match
Snort	`.*[aA][uU][tT][hH]...[iI][cC] ~ ~[^\x0A]{512}`	11.27	7.8	361.1	555.56	1.04	0.03	0.05	0.31
Snort	`\x20[^\x21\x22]{500}`	439.98	0.11	2.20	TO	1.08	0.03	0.04	0.83
Snort	`^RCPT TO\x20\s*[\w\s@\.]{200,}~ ~\x20[\w\s@\.]{200,}...`	340.7	—	TO	TO	1.68	0.03	0.07	0.89
Snort	`php.*\x20[^\n]{256}`	176.75	0.10	1.22	TO	1.08	0.04	0.07	0.74
Snort	`^(NT|CallBack|SID|TimeOut)\s*~ ~\x20\s*[^\n]{512}`	164.11	0.12	14.59	229.41	1.07	0.03	0.07	0.72
Snort	`.*[nN][eE][wW]... [^\x20]{100}`	0.13	1.26	39.92	0.74	0.81	0.03	0.04	0.65
Bro	`^[nN][aA][mM][eE]=s*[^\r\n\x3b~ ~\x20\x09\x0b\x2c]{300}`	128.57	12.24	0.51	76.48	1.15	0.03	0.04	0.94
Sw	`_.{39}`	22.96	225.34	1.94	357.68	1.12	0.03	0.04	0.79
Sw	`(.{1,980}[,])\s+(\S)`	260.59	TO	308.66	0.63	1.07	0.03	0.05	0.59
Sw	`(_a ){64999}_a`	—	—	TO	TO	0.96	0.03	0.04	0.51
Sw	`\[{50000}a\]{50000}`	—	—	4.36	TO	5.13	0.02	0.02	0.41
Sw	`^QS([NDR])(.{4})(.{6})(\d{8})...~ ~(.{4})(.{6})(.{8})(.{8})(.)$`	0.12	0.10	1.03	0.85	96.20	0.04	81.64	0.65

Regexes with nondeterministic counting are the main focus of our benchmark. Namely, they constitute 67 % of the 1,789 regexes used. From the 1,284 regexes that were at least *slightly* problematic for some of the other tools except CA (it took some tool ≥ 1 s), 73 % of them were with nondeterministic counting. From the 454 regexes that were *significantly* problematic for some of the other tools (it took some tool ≥ 10 s), 85 % of them had nondeterministic counting. From the 109 regexes that were *problematic* for *all* other tools (≥ 1 s), 100 % were with nondeterministic counting. As shown in the results above, our approach can deal with nondeterministic counting quite well.

7.1.2 Adversarial Regexes. Another ReDoS scenario is when the attacker can control the regex to be used for matching. Creating a counting regex causing efficiency problems for a given text is easier than generating adversarial texts. For instance, the regex `[a-zA-Z().,' ]*[a-zA-Z ][a-zA-Z();' ]{250}` was obtained as a modification of the running example "`.*a.{`k`}`" (where a appears k positions from the end). When run on a ~4 MiB English text with sufficiently long lines, RE2 took 86 s, grep took 26 s, while CA took only 1.1 s. Similar examples could be obtained from regexes from Section 7.1 for which some specific difficult text can be generated, namely by widening their character classes.. Our approach solves a large class of the dangerous cases, allowing one to significantly alleviate restrictions put on the user for security/efficiency reasons.

7.2 Robustness wrt Counter Values

This experiment measures the ability of the tools to cope with increasing counter bounds. For this, we selected the regex "`(_a ){`k`}_a`" where k is a parameter (the original regex `(_a ){64999}_a` comes from [?]) and measured the time the tools took on a ~500 KiB text created by our generator for increasing values of k. We give the results in Fig. 5b (the timeout was 40 s).

With the increasing value of k, the time needed by CA stays constant, around 0.35 s, while the time needed by other tools grows. In particular, .NET and SRM have cubic trends wrt the value of k, while RE2 and grep grow linearly. Notice that, for RE2 and grep, their matching time is low (around 0.01 s) until they reach a threshold from which they start behaving linearly. This corresponds to

the situation when the size of the cache for storing states of the NFA-to-DFA construction is not enough to accommodate the DFA states exercised by the input adversarial text. This yields repeated flushing of the cache, making it ineffectual.

7.3 Adversarial Text Generation

RE2 and grep store powerstates of the NFA-to-DFA construction in a cache. In typical cases, the amount of cache misses is low and almost the entire text is processed using the cache, which is extremely fast. If the cache, however, exceeds a given size, it is flushed. If the input text is such that the DFA run sees many different states, then cache misses are frequent, so large powerstates need to be constructed often, and the performance of the matching drops.

Therefore, we focus on generating texts that force exploration of many new large powerstates. In essence, we explore the configuration space of the CsA with the goal of finding as many large configurations as possible, with the focus on generating large counting sets. We partially drive the search towards loops in the CsA structure that have a potential to create large counting sets: the loops use counters with large bounds, do not contain exits, and contain Rst or Rst1 operations. For space reasons, we omit the technical details here; perfecting this method for stress testing automata-based matchers is, however, one of our future goals.

7.4 A Note on the Maturity of the Tools

The aim of our experiments is comparing algorithms rather than tools, and it should be noted that CA is much less optimized than the rest. This holds especially for RE2 and grep, which have both been actively developed for over 10 years and the amount of engineering effort invested into making them fast is substantial. The optimizations are both high-level, such as using the Boyer-Moore algorithm for skipping sections of the input text, and low-level, such as using C/C++, on-the-fly determinization, or optimizing memory accesses [**??**]. On the other hand, although there have been some optimizations done in CA (such as finding a start of a match), their nature is still quite simple. The three tools are, however, all based on the same principle of using deterministic automata, and many of the optimizations and heuristics in RE2 and grep (at least all of those mentioned above) could be directly re-applied in our setting. SRM builds on the .NET framework and reuses the .NET regex parser while replacing the built-in backtracking back-end matcher with a matching engine based on Brzozowski-style symbolic derivatives to create the DFA on the fly. In fact, CA builds on the open-source codebase of SRM and extends it with counters.

8 RELATED WORK

Regexes and their derivatives. Brzozowski derivatives [**?**] provide a practical approach to incrementally creating a DFA from a regex and can be used for efficient matching [**??**] and match generation [**?**]. Efficient determinization based on Brzozowski derivatives was first investigated in [**?**]. In the classical setting, Antimirov derivatives [**?**] are used to construct NFAs from regexes, and may in some cases result in exponentially more succinct automata than the corresponding DFAs constructed with Brzozowski derivatives. The precise connection between conditional derivatives defined in Section 5 and Antimirov derivatives is that, without counting loops, $\{D \mid \langle \mathbf{ID}, D\rangle \in \partial_a(S)\}$ is exactly the Antimirov derivative of R for a. The Antimirov construction has also been generalized to extended regexes [**?**] allowing Boolean operators such as complement and intersection. Basic theoretical properties between various automata formalisms and derivatives are discussed in [**?**].

Automata with counting. This work is a continuation of our recent work [**?**]. In [**?**], we propose a general determinization of CAs that can produce smaller automata than the naive explicit determinization but has the same worst-case complexity, which depends on the counters with the

factor $(K+1)^{|C|}$ where C is the set of counters and K the maximum counter upper bound. It also proposes a more efficient algorithm for the class of monadic regexes (single-state-scoped counters and counting on self-loops only), but it can still generate $(K+1)^{|Q|}$ states (for example, it would generate $K+1$ states for the regex from Fig. 1)—while the complexity of our determinization does not depend on K. [**?**] also present neither a derivative construction for translating regexes into CAs nor an application of CAs in pattern matching.

The use of counters has also been investigated in [**?**] for regexes with bounded repetition, building on the formalism of counter automata called CNFAs [**?**]. A CA in the current paper is essentially a symbolic generalization of a CNFA with some small technical differences, such as counters being 0-based as opposed to 1-based in a CNFA. The latter difference is mainly due to our use of a generalized *Antimirov* construction of CAs, as opposed to a generalized *Glushkov* construction used in [**?**], which is algorithmically quite different. The work in [**?**] focuses mostly on deterministic regexes and on a different problem, namely, the so-called incremental matching in the context of database queries (a query is repeatedly evaluated on a gradually changing word). For standard matching, it uses a variant of the Thompson's algorithm applied directly on a CA instead of an NFA (hence the translation of the regex to an automaton does not depend on the counter bounds, but each text character is processed with the same cost as with the original Thompson's algorithm, at worst linear to the size of the NFA and the counter bounds). This algorithm is indeed fast on deterministic regexes from practice but can slow down significantly on nondeterministic ones (which we witnessed in several experiments with the prototype implementation of [**?**] on several of our regexes).

The work in [**?**] is a theoretical study of matching regexes with counting. It proposes a matching algorithm based on dynamic programming that runs in time at worst quadratic to the length of the text (while determinization and NFA-simulation-based algorithms run in time linear to the text length). The experimental comparison of [**?**] with their variant of Thompson's algorithm suggests that the matching algorithm of [**?**] is indeed not competitive in practice.

Extended FAs (XFAs) augment classical automata with a scratch memory of bits [**??**] that can represent counters. Regexes are compiled into deterministic XFAs by first using an extended version of the Thompson's algorithm, followed by an extended version of the classical powerset construction and minimization. Although a small XFA may exist, the determinization algorithm incurs an intermediate exponential blowup of the search space for inputs such as `.*a.{`k`}`.

R-automata [**?**] are also related to our CAs, but their counters need not have upper bounds and cannot be tested or compared. Further, there are various notions of extended finite state machines whose expressive power goes beyond regular languages, e.g., [**????**]. Such automata are, however, not suitable for the problem of pattern matching considered here.

Regexes with counting. Regexes with counters are also discussed in [**???**]. The automata with counters used in [**?**], called FACs, are close to our CAs, but we allow symbolic character predicates and more kinds of counter updates. The conversion from regexes to FACs proposed in [**?**] uses a variant of Glushkov automata [**?**] and the `first-last-follow` construction [**??**]. For us, the Antimirov-derivative-based construction was easier to implement and provides benefits that are not available otherwise. Namely, it allows subsumption checking between regexes, and it generates fewer counters (one per distinct counter subexpression rather than one per counter position in the regex abstract syntax tree). While all these algorithms generate ϵ-free automata, they differ in complexity [**?**] and are thus not merely different disguises of the same technique. In particular, the Antimirov automaton is in general smaller than the Glushkov automaton with up to $n+1$ states and up to n^2 transitions. The Antimirov automaton is in fact a quotient of the Glushkov automaton [**??**]. Another generalization of Antimirov derivatives [**?**] introduces expressions kR

where R is a rational expression and k a multiplicity from a semiring such as $\mathbb{Q}$; this generalization is unrelated to counters.

An open question is whether the generalized Antimirov construction can be extended to work with Brzozowski derivatives [**?**]; we believe that such an extension, if it exists, is not straightforward because it would give rise to a direct and incremental determinization algorithm.

There are also works on regexes with counting that translate deterministic regexes to CAs and work with different notions of determinism [**??**]. A central result in [**?**] is that *counter-1-unambiguous* regexes can be compiled into deterministic FACs and that checking determinism of FACs can be done in polynomial time. The related work in [**?**] studies membership in regexes with counting. None of these papers addresses the problem of determinizing nondeterministic CAs.

Pattern matching of regexes with counting. The counting operator often appears in regexes in practice. In particular, our analysis of the 537k real-world regexes obtained in the study performed by Davis et al. [**?**] showed that over 33k regexes contained the counting operator.

GNU `grep` [**?**] (written in C) and `RE2` [**?**] (written in C++) are extremely optimized regex matchers. Both are based on translating the regex into an NFA and performing an on-the-fly determinization during the matching, avoiding a costly *a priori* determinization, while keeping a good performance by avoiding backtracking. (The translation into FAs is only allowed when the regex does not include back-references, which allow to express some context-free propeties). Both engines process the counting operator by first rewriting a regex of the form `<re>{`n,m`}` into `<re>. . . <re><re>{0,`$m-n$`}`. The regex `<re>{0,`k`}` is then transformed into `(<re>(<re>(. . . <re>?)?)?)?` (see [**?**] for more details).

In the .NET ecosystem, we are aware of two regex matchers. The first one is the standard .NET regex matcher provided in `System.Text.RegularExpressions`, which is based on a backtracking search. The other one is Symbolic Regex Matcher (SRM) of [**?**] based on the so-called *symbolic derivatives*, which provide a backtracking-free search (without an explicit conversion into a DFA) and can deal more efficiently with the counting operator.

9 CONCLUSIONS AND FUTURE WORK

We have presented a framework for efficient pattern matching of regexes with counting, which includes a derivative construction to compile regexes to counting automata, their subsequent determinization into novel counting-set automata, and a fast matching algorithm. The resources needed to build the CsAs are independent of counter bounds. It handles a majority of regexes with counting found in practice, with a much more stable performance than other matchers.

In the future, we intend to explore the limits of the idea of counting sets to enlarge and clearly delimit the class of regexes and counting automata that can be succinctly determinized while preserving fast matching. We also plan to explore possible usage of CsAs as a replacement of classical automata in other applications where automata are used, for instance, as symbolic representations of state spaces. For this, we intend to develop CsA counterparts of essential automata techniques, such as Boolean operations and minimization/size-reduction techniques. We also wish to elaborate on our method for generating texts for stress-testing matchers on regexes with counting.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and also Juraj Síč for their valuable comments and suggestions. This work is supported by the Czech Ministry of Education, Youth and Sports project LL1908 of the ERC.CZ programme, and the FIT BUT internal project FIT-S-20-6427.